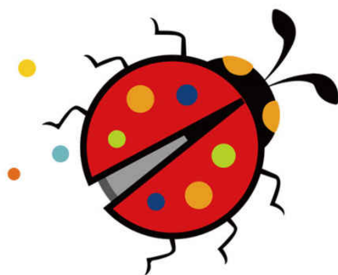
资深Python工程师10年开发经验分享

精通Scrapy网络爬虫

刘硕 编著

- 深入讲解Scrapy的核心技术与知识点
- 从简单爬虫到分布式爬虫，涉及面广泛
- 注重实践，范例丰富，代码注释详尽



清華大学出版社

VISIT...

LANZAROTE
Caliente.com

资深Python工程师10年开发经验分享

精通Scrapy网络爬虫

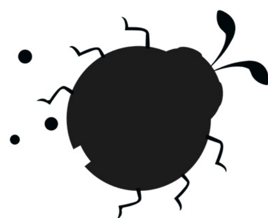
刘硕 编著

- 深入讲解Scrapy的核心技术与知识点
- 从简单爬虫到分布式爬虫，涉及面广泛
- 注重实践，范例丰富，代码注释详尽



清华大学出版社

精通Scrapy网络爬虫



本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：**010-62782989 13701121933**

图书在版编目（**CIP**）数据

精通Scrapy网络爬虫／刘硕编著．—北京：清华大学出版社，2017

ISBN 978-7-302-48493-6

Ⅰ．①精…　Ⅱ．①刘…　Ⅲ．①软件工具－程序设计　Ⅳ．①TP311.561

中国版本图书馆CIP数据核字（2017）第230263号

责任编辑：王金柱

封面设计：王　翔

责任校对：闫秀华

责任印制：

出版发行：清华大学出版社

网　　址：http://www.tup.com.cn，http://www.wqbook.com

地　　址：北京清华大学学研大厦A座

邮　　编：100084

社总机：010-62770175

邮　　购：010-62786544

投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn

质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印装者：

经　　销：全国新华书店

开　　本：180mm×230mm

印　　张：14.5

字　　数：325千字

版　　次：2017年10月第1版

印　　次：2017年10月第1次印刷

印　　数：1～3000

定　　价：59.00元

产品编号：074439-01

内容简介

本书深入系统地介绍了Python流行框架Scrapy的相关技术及使用技巧。全书共14章，从逻辑上可分为基础篇和高级篇两部分，基础篇重点介绍Scrapy的核心元素，如spider、selector、item、link等；高级篇讲解爬虫的高级话题，如登录认证、文件下载、执行JavaScript、动态网页爬取、使用HTTP代理、分布式爬虫的编写等，并配合项目案例讲解，包括供练习使用的网站，以及京东、知乎、豆瓣、360爬虫案例等。

本书案例丰富，注重实践，代码注释详尽，适合有一定Python语言基础，想学习编写复杂网络爬虫的读者使用。

作者简介

刘硕：硕士，曾就职于知名外企，从事一线开发工作10年，目前主要从事Python开发与教学工作，在慕课网开设有多门Python课程，深受学员欢迎。

前言

关于本书

如今是互联网的时代，而且正在迈入智能时代。人们早已意识到互联网中的数据是有待开采的巨大金矿，这些数据将会改善我们的生活，网络爬虫开发工作岗位的出现和不断增加正是基于对数据价值的重视。优秀的爬虫框架就像是开采金矿的强力挖掘机，如果你能娴熟地驾驶它们，就能大幅提高开采效率。

本书讲解目前最流行的Python爬虫框架Scrapy，它简单易用、灵活易拓展、文档丰富、开发社区活跃，使用Scrapy可以高效地开发网络爬虫应用。本书的读者只需要有Python语言基础即可，我们从零基础、逐步由浅入深进行讲解。第1～8章讲解Scrapy开发的核心基础部分，其中包括：

- 初识Scrapy
- 编写Spider
- 使用Selector提取数据
- 使用Item封装数据
- 使用Item Pipeline处理数据
- 使用Link Extractor提取链接
- 使用Exporter导出数据
- 项目练习

第9～14章讲解实际爬虫开发中使用频率最高的一些实用技术，其中包括：

- 下载文件和图片
- 模拟登录
- 爬取动态页面

- 存入数据库
- 使用HTTP代理
- 分布式爬取

本书特色

本书的宗旨是以实用和实战为教学目标，主要特色是：

- 所有基础部分的讲解都配有代码示例，而不仅仅是枯燥的文档。
- 案例选材方面以讲解知识点为核心，尽量选择专门供练习爬虫技术的网站（不易变动）或贴近日常生活的网站（京东、知乎、豆瓣、360）进行演示。
- 在讲解某些知识点时，对Scrapy源码进行分析，让读者能够“知其然并知其所以然”。

另外，Python是一门简单易学、功能强大、开发效率极高的语言，近年来在网络爬虫、数据分析、机器学习等领域得到广泛认可。虽然Python很容易上手，但想灵活恰当地运用它也并不简单。作者在慕课网（www.imooc.com）上推出了一套《Python高级进阶实战》课程，可供有需求的读者进行参考：http://coding.imooc.com/class/62.html。

致谢

感谢康烁和陈渝老师在清华大学信息研究院工作期间对我在专业方面的耐心指导。

感谢清华大学出版社的王金柱编辑给予我这次写作的机会以及在写作方面的指点。

感谢赵佳音同事认真阅读全书并提出了许多的宝贵建议。

感谢剑超和任怡同学认真审阅全书并对书中代码在多个Python版本上进行测试。

感谢女儿刘真，她的笑容化解了写作本书时偶尔的小烦躁。

编　者

2017年8月8日

目 录

前言
第1章　初识Scrapy
1.1　网络爬虫是什么
1.2　Scrapy简介及安装
1.3　编写第一个Scrapy爬虫
1.3.1　项目需求
1.3.2　创建项目
1.3.3　分析页面
1.3.4　实现Spider
1.3.5　运行爬虫
1.4　本章小结
第2章　编写Spider
2.1　Scrapy框架结构及工作原理
2.2　Request和Response对象
2.2.1　Request对象
2.2.2　Response对象
2.3　Spider开发流程
2.3.1　继承scrapy.Spider
2.3.2　为Spider命名
2.3.3　设定起始爬取点
2.3.4　实现页面解析函数
2.4　本章小结
第3章　使用Selector提取数据
3.1　Selector对象
3.1.1　创建对象
3.1.2　选中数据
3.1.3　提取数据
3.2　Response内置Selector
3.3　XPath
3.3.1　基础语法
3.3.2　常用函数

3.4　CSS选择器
3.5　本章小结
第4章　使用Item封装数据
4.1　Item和Field
4.2　拓展Item子类
4.3　Field元数据
4.4　本章小结
第5章　使用Item Pipeline处理数据
5.1　Item Pipeline
5.1.1　实现Item Pipeline
5.1.2　启用Item Pipeline
5.2　更多例子
5.2.1　过滤重复数据
5.2.2　将数据存入MongoDB
5.3　本章小结
第6章　使用LinkExtractor提取链接
6.1　使用LinkExtractor
6.2　描述提取规则
6.3　本章小结
第7章　使用Exporter导出数据
7.1　指定如何导出数据
7.1.1　命令行参数
7.1.2　配置文件
7.2　添加导出数据格式
7.2.1　源码参考
7.2.2　实现Exporter
7.3　本章小结
第8章　项目练习
8.1　项目需求
8.2　页面分析
8.3　编码实现
8.4　本章小结
第9章　下载文件和图片
9.1　FilesPipeline和ImagesPipeline

9.1.1　FilesPipeline使用说明
9.1.2　ImagesPipeline使用说明
9.2　项目实战：爬取matplotlib例子源码文件
9.2.1　项目需求
9.2.2　页面分析
9.2.3　编码实现
9.3　项目实战：下载360图片
9.3.1　项目需求
9.3.2　页面分析
9.3.3　编码实现
9.4　本章小结
第10章　模拟登录
10.1　登录实质
10.2　Scrapy模拟登录
10.2.1　使用FormRequest
10.2.2　实现登录Spider
10.3　识别验证码
10.3.1　OCR识别
10.3.2　网络平台识别
10.3.3　人工识别
10.4　Cookie登录
10.4.1　获取浏览器Cookie
10.4.2　CookiesMiddleware源码分析
10.4.3　实现BrowserCookiesMiddleware
10.4.4　爬取知乎个人信息
10.5　本章小结
第11章　爬取动态页面
11.1　Splash渲染引擎
11.1.1　render.html端点
11.1.2　execute端点
11.2　在Scrapy中使用Splash
11.3　项目实战：爬取toscrape中的名人名言
11.3.1　项目需求
11.3.2　页面分析

11.3.3　编码实现
11.4　项目实战：爬取京东商城中的书籍信息
11.4.1　项目需求
11.4.2　页面分析
11.4.3　编码实现
11.5　本章小结
第12章　存入数据库
12.1　SQLite
12.2　MySQL
12.3　MongoDB
12.4　Redis
12.5　本章小结
第13章　使用HTTP代理
13.1　HttpProxyMiddleware
13.1.1　使用简介
13.1.2　源码分析
13.2　使用多个代理
13.3　获取免费代理
13.4　实现随机代理
13.5　项目实战：爬取豆瓣电影信息
13.5.1　项目需求
13.5.2　页面分析
13.5.3　编码实现
13.6　本章小结
第14章　分布式爬取
14.1　Redis的使用
14.1.1　安装Redis
14.1.2　Redis基本命令
14.1.3　Python访问Redis
14.2　scrapy-redis源码分析
14.2.1　分配爬取任务部分
14.2.2　汇总爬取数据部分
14.3　使用scrapy-redis进行分布式爬取
14.3.1　搭建环境

14.3.2　项目实战
14.4　本章小结

第1章 初识Scrapy

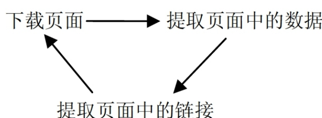
本章首先介绍爬虫的基本概念、工作流程，然后介绍Scrapy的安装和网络爬虫项目的实现流程，使读者对网络爬虫有一个大致的了解，并且建立起网络爬虫的编写思路。本章重点讲解以下内容：

- 网络爬虫及爬虫的工作流程。
- Scrapy的介绍与安装。
- 网络爬虫编写步骤。

1.1 网络爬虫是什么

网络爬虫是指在互联网上自动爬取网站内容信息的程序，也被称作网络蜘蛛或网络机器人。大型的爬虫程序被广泛应用于搜索引擎、数据挖掘等领域，个人用户或企业也可以利用爬虫收集对自身有价值的数据。举一个简单的例子，假设你在本地新开了一家以外卖生意为主的餐馆，现在要给菜品定价，此时便可以开发一个爬虫程序，在美团、饿了么、百度外卖这些外卖网站爬取大量其他餐馆的菜品价格作为参考，以指导定价。

一个网络爬虫程序的基本执行流程可以总结为以下循环：



1．下载页面

一个网页的内容本质上就是一个HTML文本，爬取一个网页内容之前，首先要根据网页的URL下载网页。

2．提取页面中的数据

当一个网页（HTML）下载完成后，对页面中的内容进行分析，并提取出我们感兴趣的数据，提取到的数据可以以多种形式保存起来，比如将数据以某种格式（CSV、JSON）写入文件中，或存储到数据库（MySQL、MongoDB）中。

3．提取页面中的链接

通常，我们想要获取的数据并不只在一个页面中，而是分布在多个页面中，这些页面彼此联系，一个页面中可能包含一个或多个到其他页面的链接，提取完当前页面中的数据后，还要把页面中的某些链接也提取出来，然后对链接页面进行爬取（循环1-3步骤）。

设计爬虫程序时，还要考虑防止重复爬取相同页面（URL去重）、网页搜索策略（深度优先或广度优先等）、爬虫访问边界限定等一系列问题。

从头开发一个爬虫程序是一项烦琐的工作，为了避免因制造轮子而消耗大量时间，在实际应用中我们可以选择使用一些优秀的爬虫框架，使用框架可以降低开发成本，提高程序质量，让我们能够专注于业务逻辑（爬取有价值的数据）。接下来，本书就带你学习目前非常流行的开源爬虫框架Scrapy。

1.2　Scrapy简介及安装

Scrapy是一个使用Python语言（基于Twisted框架）编写的开源网络爬虫框架，目前由Scrapinghub Ltd维护。Scrapy简单易用、灵活易拓展、开发社区活跃，并且是跨平台的。在Linux、　MaxOS以及Windows平台都可以使用。Scrapy应用程序也使用Python进行开发，目前可以支持Python 2.7以及Python 3.4＋版本。

在任意操作系统下，可以使用pip安装Scrapy，例如：

```
$ pip install scrapy
```

为确认Scrapy已安装成功，首先在Python中测试能否导入Scrapy模块：

```
>>> import scrapy
>>> scrapy.version_info
(1, 3, 3)
```

然后，在shell中测试能否执行Scrapy这条命令：

```
$ scrapy
Scrapy 1.3.3 - no active project


Usage:
 scrapy [options] [args]


Available commands:
 bench          Run quick benchmark test
 commands
 fetch         Fetch a URL using the Scrapy downloa
 genspider     Generate new spider using pre-define
 runspider     Run a self-contained spider (without
 settings     Get settings values
 shell         Interactive scraping console
 startproject Create new project
 version       Print Scrapy version
 view           Open URL in browser, as seen by Scr


  [ more ]      More commands available when run fro


Use "scrapy -h" to see more info about a command
```

通过了以上两项检测，说明Scrapy安装成功了。如上所示，我们

安装的是当前最新版本1.3.3。

1.3 编写第一个Scrapy爬虫

为了帮助大家建立对Scrapy框架的初步印象，我们使用它完成一个简单的爬虫项目。

1.3.1 项目需求

在专门供爬虫初学者训练爬虫技术的网站（http://books.toscrape.com）上爬取书籍信息，如图1-1所示。

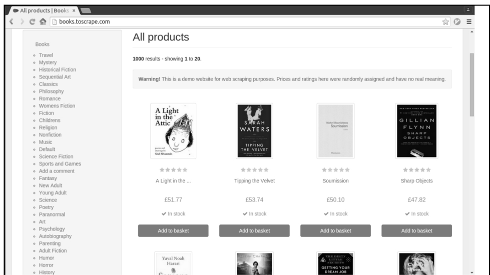


图1-1

该网站中，这样的书籍列表页面一共有50页，每页有20本书，第一个例子应尽量简单。我们下面仅爬取所有图书（1000本）的书名和价格信息。

1.3.2 创建项目

首先，我们要创建一个Scrapy项目，在shell中使用scrapy startproject命令：

```
$ scrapy startproject example
New Scrapy project 'example', using template direc
'/usr/local/lib/python3.4/dist-packages/scrapy/tem
```

```
/home/liushuo/book/example

You can start your first spider with:
cd example
scrapy genspider example example.com
```

创建好一个名为example的项目后，可使用tree命令查看项目目录下的文件，显示如下：

```
$ tree example
example/
├── example
│   ├── __init__.py
│   ├── items.py
│   ├── middlewares.py
│   ├── pipelines.py
│   ├── settings.py
│   └── spiders
│       └── __init__.py
└── scrapy.cfg
```

随着后面逐步深入学习，大家会了解这些文件的用途，此处不做解释。

1.3.3　分析页面

编写爬虫程序之前，首先需要对待爬取的页面进行分析，主流的浏览器中都带有分析页面的工具或插件，这里我们选用Chrome浏览器的开发者工具（Tools→Developer tools）分析页面。

1．数据信息

在Chrome浏览器中打开页面http://books.toscrape.com，选中其中任意一本书并右击，然后选择“审查元素”，查看其HTML代码，如图1-2所示。

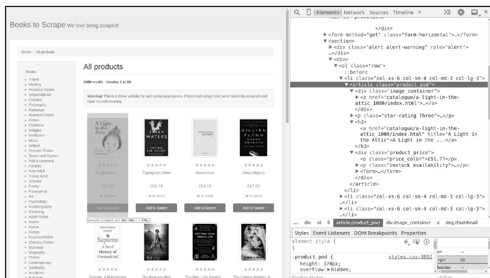


图1-2

可以看到，每一本书的信息包裹在<article class="product_pod">元素中：书名信息在其下h3 > a元素的title属性中，如A Light in the...；书价信息在其下<p class="price_color">元素的文本中，如<p class="price_color">£51.77</p>。

2．链接信息

图1-3所示为第一页书籍列表页面，可以通过单击next按钮访问下一页，选中页面下方的next按钮并右击，然后选择“审查元素”，查看其HTML代码，如图1-3所示。

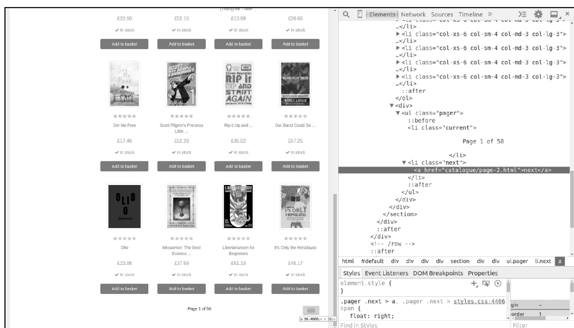


图1-3

可以发现，下一页的URL在ul.pager > li.next > a元素的href属性中，是一个相对URL地址，如<li class="next">next。

1.3.4 实现Spider

分析完页面后，接下来编写爬虫。在Scrapy中编写一个爬虫，即实现一个scrapy.Spider的子类。

实现爬虫的Python文件应位于exmaple/spiders目录下，在该目录下创建新文件book_spider.py。然后，在book_spider.py中实现爬虫BooksSpider，代码如下：

```
# -*- coding: utf-8 -*-
import scrapy


class BooksSpider(scrapy.Spider):
   # 每一个爬虫的唯一标识
   name = "books"


   # 定义爬虫爬取的起始点，起始点可以是多个，这里只有一个
   start_urls = ['http://books.toscrape.com/']


   def parse(self, response):
       # 提取数据
       # 每一本书的信息在<article class="product_pod"
       # css()方法找到所有这样的article 元素，并依次迭代
       for book in response.css('article.product_p
           # 书名信息在article > h3 > a 元素的title属
           # 例如：<a title="A Light in the Attic">
           name = book.xpath('./h3/a/@title').extr


           # 书价信息在 <p class="price_color">的TEX
           # 例如：<p class="price_color">£51.77</
           price = book.css('p.price_color::text')
           yield {
              'name': name,
              'price': price,
```

```
            }

        # 提取链接
        # 下一页的url 在ul.pager > li.next > a 里面
        # 例如: <li class="next"><a href="catalogue/
        next_url = response.css('ul.pager li.next a
        if next_url:
            # 如果找到下一页的URL，得到绝对路径，构造新的R
            next_url = response.urljoin(next_url)
            yield scrapy.Request(next_url, callback
```

如果上述代码中有看不懂的部分，大家不必担心，更多详细内容会在后面章节学习，这里只要先对实现一个爬虫有个整体印象即可。

下面对BooksSpider的实现做简单说明。

- name属性

  一个Scrapy项目中可能有多个爬虫，每个爬虫的name属性是其自身的唯一标识，在一个项目中不能有同名的爬虫，本例中的爬虫取名为'books'。

- start_urls属性

  一个爬虫总要从某个（或某些）页面开始爬取，我们称这样的页面为起始爬取点，start_urls属性用来设置一个爬虫的起始爬取点。在本例中只有一个起始爬取点'http://books.toscrape.com'。

- parse方法

  当一个页面下载完成后，Scrapy引擎会回调一个我们指定的页面解析函数（默认为parse方法）解析页面。一个页面解析函数通常需要完成以下两个任务：

  - 提取页面中的数据（使用XPath或CSS选择器）。
  - 提取页面中的链接，并产生对链接页面的下载请求。

页面解析函数通常被实现成一个生成器函数，每一项从页面中提

取的数据以及每一个对链接页面的下载请求都由yield语句提交给Scrapy引擎。

1.3.5 运行爬虫

完成代码后，运行爬虫爬取数据，在shell中执行scrapy crawl <SPIDER_NAME>命令运行爬虫'books'，并将爬取的数据存储到csv文件中：

```
     $ scrapy crawl books -o books.csv
     2016-12-27 15:19:53 [scrapy] INFO: Scrapy 1.3
     2016-12-27 15:19:53 [scrapy] INFO: INFO: Over
'SPIDER_MODULES': ['example.spiders'], 'ROBOTSTXT_
'example.spiders'}
     2016-12-27 15:19:53 [scrapy] INFO: Enabled ex
     ['scrapy.extensions.telnet.TelnetConsole',
      'scrapy.extensions.corestats.CoreStats',
      'scrapy.extensions.feedexport.FeedExporter',
      'scrapy.extensions.logstats.LogStats']
     2016-12-27 15:19:53 [scrapy] INFO: Enabled do
     ['scrapy.downloadermiddlewares.robotstxt.Robo
      'scrapy.downloadermiddlewares.httpauth.HttpA
      'scrapy.downloadermiddlewares.downloadtimeou
      'scrapy.downloadermiddlewares.defaultheaders
      'scrapy.downloadermiddlewares.useragent.User
      'scrapy.downloadermiddlewares.retry.RetryMid
      'scrapy.downloadermiddlewares.redirect.MetaR
      'scrapy.downloadermiddlewares.httpcompressio
      'scrapy.downloadermiddlewares.redirect.Redir
      'scrapy.downloadermiddlewares.cookies.Cookie
      'scrapy.downloadermiddlewares.chunked.Chunke
      'scrapy.downloadermiddlewares.stats.Download
     2016-12-27 15:19:53 [scrapy] INFO: Enabled sp
     ['scrapy.spidermiddlewares.httperror.HttpErro
      'scrapy.spidermiddlewares.offsite.OffsiteMid
      'scrapy.spidermiddlewares.referer.RefererMid
      'scrapy.spidermiddlewares.urllength.UrlLengt
```

```
     'scrapy.spidermiddlewares.depth.DepthMiddlew
    2016-12-27 15:19:53 [scrapy] INFO: Enabled it
    []
    2016-12-27 15:19:53 [scrapy] INFO: Spider ope
    2016-12-27 15:19:53 [scrapy] INFO: Crawled 0
items/min)
    2016-12-27 15:19:53 [scrapy] DEBUG: Telnet co
    2016-12-27 15:20:01 [scrapy] DEBUG: Crawled (
    2016-12-27 15:20:02 [scrapy] DEBUG: Crawled (
    2016-12-27 15:20:02 [scrapy] DEBUG: Scraped f
    {'name': 'A Light in the Attic', 'price': '£
    2016-12-27 15:20:02 [scrapy] DEBUG: Scraped f
    {'name': 'Tipping the Velvet', 'price': '£53
    2016-12-27 15:20:02 [scrapy] DEBUG: Scraped f
    {'name': 'Soumission', 'price': '£50.10'}


    ... <省略中间部分输出> ...


    2016-12-27 15:21:30 [scrapy] DEBUG: Scraped f
http://books.toscrape.com/catalogue/page-50.html>
    {'name': '1,000 Places to See Before You Die'
    2016-12-27 15:21:30 [scrapy] INFO: Closing sp
    2016-12-27 15:21:30 [scrapy] INFO: Stored csv
    2016-12-27 15:21:30 [scrapy] INFO: Dumping Sc
    {'downloader/request_bytes': 14957,
     'downloader/request_count': 51,
     'downloader/request_method_count/GET': 51,
 'downloader/response_bytes': 299924,
 'downloader/response_count': 51,
 'downloader/response_status_count/200': 50,
 'downloader/response_status_count/404': 1,
 'finish_reason': 'finished',
 'finish_time': datetime.datetime(2016, 12, 27, 7,
 'item_scraped_count': 1000,
 'log_count/DEBUG': 1052,
 'log_count/INFO': 9,
 'request_depth_max': 49,
 'response_received_count': 51,
```

```
 'scheduler/dequeued': 50,
 'scheduler/dequeued/memory': 50,
 'scheduler/enqueued': 50,
 'scheduler/enqueued/memory': 50,
 'start_time': datetime.datetime(2016, 12, 27, 7,
2016-12-27 15:21:30 [scrapy] INFO: Spider closed (
```

等待爬虫运行结束后，在books.csv文件中查看爬取到的数据，代码如下：

```
$ sed -n '2,$p' books.csv | cat -n      # 不显示第一
     1   A Light in the Attic,£51.77
     2   Tipping the Velvet,£53.74
     3   Soumission,£50.10
     4   Sharp Objects,£47.82
     5   Sapiens: A Brief History of Humankind,£54
     6   The Requiem Red,£22.65
     7   The Dirty Little Secrets of Getting Your D


   ... <省略中间部分输出> ...


   995   Beyond Good and Evil,£43.38
   996   Alice in Wonderland (Alice's Adventures in
   997   "Ajin: Demi-Human, Volume 1 (Ajin: Demi-Hu
   998   A Spy's Devotion (The Regency Spies of Lon
   999   1st to Die (Women's Murder Club #1),£53.9
  1000  "1,000 Places to See Before You Die",£26.0
```

从上述数据可以看出，我们成功地爬取到了1000本书的书名和价格信息（50页，每页20项）。

1.4　本章小结

本章是开始Scrapy爬虫之旅的第1章，先带大家了解了什么是网

络爬虫，然后对Scrapy爬虫框架做了简单介绍，最后以一个简单的爬虫项目让大家对开发Scrapy爬虫有了初步的印象。在接下来的章节中，我们将深入学习开发Scrapy爬虫的核心基础内容。

第2章 编写Spider

从本章开始介绍Scrapy爬虫的基础知识部分，我们先从Scrapy爬虫程序中最核心的组件Spider讲起，在学习编写Spider之前，需要一些知识作为铺垫。本章重点讲解以下内容：

- Scrapy框架结构及工作原理。
- Request对象和Response对象。

2.1 Scrapy框架结构及工作原理

图2-1展示了Scrapy框架的组成结构，并从数据流的角度揭示Scrapy的工作原理。

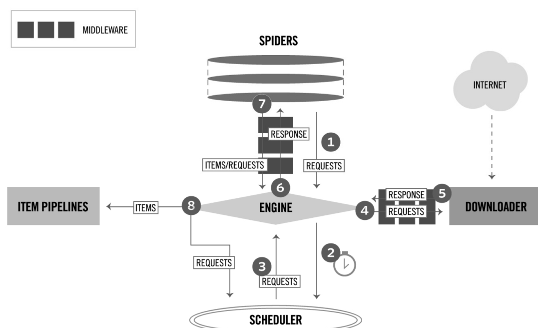


图2-1

首先，简单了解一下Scrapy框架中的各个组件，如表2-1所示。

表2-1

组　件	描　述	类　型
ENGINE	引擎，框架的核心，其他所有组件在其控制下协同工作	内部组件
SCHEDULER	调度器，负责对 SPIDER 提交的下载请求进行调度	内部组件
DOWNLOADER	下载器，负责下载页面（发送 HTTP 请求/接收 HTTP 响应）	内部组件
SPIDER	爬虫，负责提取页面中的数据，并产生对新页面的下载请求	用户实现
MIDDLEWARE	中间件，负责对 Request 对象和 Response 对象进行处理	可选组件
ITEM PIPELINE	数据管道，负责对爬取到的数据进行处理	可选组件

对于用户来说，Spider是最核心的组件，Scrapy爬虫开发是围绕实现Spider展开的。

接下来，看一下在框架中的数据流，有表2-2所示的3种对象。

表2-2

对　象	描　述
REQUEST	Scrapy 中的 HTTP 请求对象
RESPONSE	Scrapy 中的 HTTP 响应对象
ITEM	从页面中爬取的一项数据

Request和Response是HTTP协议中的术语，即HTTP请求和HTTP响应，Scrapy框架中定义了相应的Request和Response类，这里的Item代表Spider从页面中爬取的一项数据。

最后，我们来说明以上几种对象在框架中的流动过程。

- 当SPIDER要爬取某URL地址的页面时，需使用该URL构造一个Request对象，提交给ENGINE（图2-1中的1）。
- Request对象随后进入SCHEDULER按某种算法进行排队，之后的某个时刻SCHEDULER将其出队，送往DOWNLOADER（图2-1中的2、3、4）。
- DOWNLOADER根据Request对象中的URL地址发送一次HTTP请求到网站服务器，之后用服务器返回的HTTP响应构造出一个Response对象，其中包含页面的HTML文本（图2-1中的5）。
- Response对象最终会被递送给SPIDER的页面解析函数（构造Request对象时指定）进行处理，页面解析函数从页面中提取数据，封装成Item后提交给ENGINE，Item之后被送往

ITEM PIPELINES进行处理，最终可能由EXPORTER（图2-1中没有显示）以某种数据格式写入文件（csv，json）；另一方面，页面解析函数还从页面中提取链接（URL），构造出新的Request对象提交给ENGINE（图2-1中的6、7、8）。

理解了框架中的数据流，也就理解了Scrapy爬虫的工作原理。如果把框架中的组件比作人体的各个器官，Request和Response对象便是血液，Item则是代谢产物。

2.2 Request和Response对象

通过上述讲解，大家已经了解了Request和Response对象在Scrapy框架中的重要性，下面详细介绍这两个对象。

2.2.1 Request对象

Request对象用来描述一个HTTP请求，下面是其构造器方法的参数列表：

```
Request(url[, callback, method='GET', headers, bod
                encoding='utf-8', priority=0, dc
```

下面依次介绍这些参数。

- url（必选）

  请求页面的url地址，bytes或str类型，如'http://www.python.org/doc'。

- callback

  页面解析函数， Callable类型，Request对象请求的页面下载完成后，由该参数指定的页面解析函数被调用。如果未传递该参数，默认调用Spider的parse方法。

- method

HTTP请求的方法，默认为'GET'。

- headers

  HTTP请求的头部字典，dict类型，例如{'Accept': 'text/html', 'User-Agent':Mozilla/5.0'}。如果其中某项的值为None，就表示不发送该项HTTP头部，例如{'Cookie': None}，禁止发送Cookie。

- body

  HTTP请求的正文，bytes或str类型。

- cookies

  Cookie信息字典，dict类型，例如{'currency': 'USD', 'country': 'UY'}。

- meta

  Request的元数据字典，dict类型，用于给框架中其他组件传递信息，比如中间件Item Pipeline。其他组件可以使用Request对象的meta属性访问该元数据字典（request.meta），也用于给响应处理函数传递信息，详见Response的meta属性。

- encoding

  url和body参数的编码默认为'utf-8'。如果传入的url或body参数是str类型，就使用该参数进行编码。

- priority

  请求的优先级默认值为0，优先级高的请求优先下载。

- dont_filter

  默认情况下（dont_filter=False），对同一个url地址多次提交下载请求，后面的请求会被去重过滤器过滤（避免重复下载）。如果将该参数置为True，可以使请求避免被过滤，强制下载。例如，在多次爬取一个内容随时间而变化的页面时

（每次使用相同的url），可以将该参数置为True。

- errback

  请求出现异常或者出现HTTP错误时（如404页面不存在）的回调函数。

虽然参数很多，但除了url参数外，其他都带有默认值。在构造Request对象时，通常我们只需传递一个url参数或再加一个callback参数，其他使用默认值即可，代码如下：

```
>>> import scrapy
>>> request = scrapy.Request('http://books.toscrap
>>> request2 = scrapy.Request('http://quotes.toscr
```

在实际应用中，我们几乎只调用Request的构造器创建对象，但也可以根据需求访问Request对象的属性，常用的有以下几个：

- url
- method
- headers
- body
- meta

这些属性和构造器参数相对应，这里不再重复解释。

2.2.2　Response对象

Response对象用来描述一个HTTP响应，Response只是一个基类，根据响应内容的不同有如下子类：

- TextResponse
- HtmlResponse
- XmlResponse

当一个页面下载完成时，下载器依据HTTP响应头部中的Content-Type信息创建某个Response的子类对象。我们通常爬取的网页，其内容是HTML文本，创建的便是HtmlResponse对象，其中HtmlResponse和XmlResponse是TextResponse的子类。实际上，这3个子类只有细微的差别，这里以HtmlResponse为例进行讲解。

下面介绍HtmlResponse对象的属性及方法。

- url

  HTTP响应的url地址，str类型。

- status

  HTTP响应的状态码，int类型，例如200，404。

- headers

  HTTP响应的头头部，类字典类型，可以调用get或getlist方法对其进行访问，例如：

```
response.headers.get('Content-Type')
response.headers.getlist('Set-Cookie')
```

- body

  HTTP响应正文，bytes类型。

- text

  文本形式的HTTP响应正文，str类型，它是由response.body使用response.encoding解码得到的，即

```
reponse.text = response.body.decode(response.encod
```

- encoding

  HTTP响应正文的编码，它的值可能是从HTTP响应头部或正

文中解析出来的。

- request

  产生该HTTP响应的Request对象。

- meta

  即response.request.meta，在构造Request对象时，可将要传递给响应处理函数的信息通过meta参数传入；响应处理函数处理响应时，通过response.meta将信息取出。

- selector

  Selector对象用于在Response中提取数据（选择器相关话题在后面章节详细讲解）。

- xpath(query)

  使用XPath选择器在Response中提取数据，实际上它是response.selector.xpath方法的快捷方式（选择器相关话题在后面章节详细讲解）。

- css(query)

  使用CSS选择器在Response中提取数据，实际上它是response.selector.css方法的快捷方式（选择器相关话题在后面章节详细讲解）。

- urljoin（url）

  用于构造绝对url。当传入的url参数是一个相对地址时，根据response.url计算出相应的绝对url。例如，response.url为http://www.example.com/a，url为b/index.html，调用response.urljoin(url)的结果为http://www.example.com/a/b/index.html。

虽然HtmlResponse对象有很多属性，但最常用的是以下的3个方法：

- xpath(query)

- css(query)
- urljoin(url)

前两个方法用于提取数据，后一个方法用于构造绝对url。

2.3 Spider开发流程

有了前面知识的铺垫，现在回到本章的主题“编写Spider”。实现一个Spider子类的过程很像是完成一系列填空题，Scrapy框架提出以下问题让用户在Spider子类中作答：

- 爬虫从哪个或哪些页面开始爬取？
- 对于一个已下载的页面，提取其中的哪些数据？
- 爬取完当前页面后，接下来爬取哪个或哪些页面？

上面问题的答案包含了一个爬虫最重要的逻辑，回答了这些问题，一个爬虫也就开发出来了。

接下来，我们以上一章exmaple项目中的BooksSpider为例，讲解一个Spider的开发流程。为方便阅读，再次给出BooksSpider的代码：

```
# -*- coding: utf-8 -*-
import scrapy


class BooksSpider(scrapy.Spider):
    # 每一个爬虫的唯一标识
    name = "books"


    # 定义爬虫爬取的起始点，起始点可以是多个，我们这里是一个
    start_urls = ['http://books.toscrape.com/']


    def parse(self, response):
        # 提取数据
```

```
        # 每一本书的信息是在<article class="product_pod
        # css()方法找到所有这样的article 元素，并依次迭代
        for book in response.css('article.product_po
            # 书名信息在article > h3 > a 元素的title属性
    # 例如: <a title="A Light in the Attic">A Light
    name = book.xpath('./h3/a/@title').extract_firs


    # 书价信息在 <p class="price_color">的TEXT中。
    # 例如: <p class="price_color">£51.77</p>
    price = book.css('p.price_color::text').extract
    yield {
        'name': name,
        'price': price,
    }


# 提取链接
# 下一页的url 在ul.pager > li.next > a 里面
# 例如: <li class="next"><a href="catalogue/page-2.
next_url = response.css('ul.pager li.next a::attr(
if next_url:
    # 如果找到下一页的url，得到绝对路径，构造新的Request 对
    next_url = response.urljoin(next_url)
    yield scrapy.Request(next_url, callback=self.pa
```

实现一个Spider只需要完成下面4个步骤：

步骤 **01** 继承scrapy.Spider。

步骤 **02** 为Spider取名。

步骤 **03** 设定起始爬取点。

步骤 **04** 实现页面解析函数。

2.3.1 继承scrapy.Spider

Scrapy框架提供了一个Spider基类，我们编写的Spider需要继承它：

```
import scrapy
class BooksSpider(scrapy.Spider):
    ...
```

这个Spider基类实现了以下内容：

- 供Scrapy引擎调用的接口，例如用来创建Spider实例的类方法from_crawler。
- 供用户使用的实用工具函数，例如可以调用log方法将调试信息输出到日志。
- 供用户访问的属性，例如可以通过settings属性访问配置文件中的配置。

实际上，在初学Scrapy时，不必关心Spider基类的这些细节，未来有需求时再去查阅文档即可。

2.3.2 为Spider命名

在一个Scrapy项目中可以实现多个Spider，每个Spider需要有一个能够区分彼此的唯一标识，Spider的类属性name便是这个唯一标识。

```
class BooksSpider(scrapy.Spider):
    name = "books"
    ...
```

执行scrapy crawl命令时就用到了这个标识，告诉Scrapy使用哪个Spider进行爬取。

2.3.3 设定起始爬取点

Spider必然要从某个或某些页面开始爬取，我们称这些页面为起

始爬取点，可以通过类属性start_urls来设定起始爬取点：

```
class BooksSpider(scrapy.Spider):
    ...
    start_urls = ['http://books.toscrape.com/']
    ...
```

start_urls通常被实现成一个列表，其中放入所有起始爬取点的url（例子中只有一个起始点）。看到这里，大家可能会想，请求页面下载不是一定要提交Request对象么？而我们仅定义了url列表，是谁暗中构造并提交了相应的Request对象呢？通过阅读Spider基类的源码可以找到答案，相关代码如下：

```
class Spider(object_ref):
    ...
    def start_requests(self):
        for url in self.start_urls:
            yield self.make_requests_from_url(url)


    def make_requests_from_url(self, url):
        return Request(url, dont_filter=True)


    def parse(self, response):
        raise NotImplementedError
    ...
```

从代码中可以看出，Spider基类的start_requests方法帮助我们构造并提交了Request对象，对其中的原理做如下解释：

- 实际上，对于起始爬取点的下载请求是由Scrapy引擎调用Spider对象的start_requests方法提交的，由于BooksSpider类没有实现start_requests方法，因此引擎会调用Spider基类的start_requests方法。

- 在start_requests方法中，self.start_urls便是我们定义的起始爬取点列表（通过实例访问类属性），对其进行迭代，用迭代出的每个url作为参数调用make_requests_from_url方法。
- 在make_requests_from_url方法中，我们找到了真正构造Reqeust对象的代码，仅使用url和dont_filter参数构造Request对象。
- 由于构造Request对象时并没有传递callback参数来指定页面解析函数，因此默认将parse方法作为页面解析函数。此时BooksSpider必须实现parse方法，否则就会调用Spider基类的parse方法，从而抛出NotImplementedError异常（可以看作基类定义了一个抽象接口）。
- 起始爬取点可能有多个，start_requests方法需要返回一个可迭代对象（列表、生成器等），其中每一个元素是一个Request对象。这里，start_requests方法被实现成一个生成器函数（生成器对象是可迭代的），每次由yield语句返回一个Request对象。

由于起始爬取点的下载请求是由引擎调用Spider对象的start_requests方法产生的，因此我们也可以在BooksSpider中实现start_requests方法（覆盖基类Spider的start_requests方法），直接构造并提交起始爬取点的Request对象。在某些场景下使用这种方式更加灵活，例如有时想为Request添加特定的HTTP请求头部，或想为Request指定特定的页面解析函数。

以下是通过实现start_requests方法定义起始爬取点的示例代码（改写BooksSpider）：

```
class BooksSpider(scrapy.Spider):

    # start_urls = ['http://books.toscrape.com/']
```

```
        # 实现start_requests 方法, 替代start_urls类属性
        def start_requests(self):
            yield scrapy.Request('http://books.toscrape
                                 callback=self.parse_b
                                 headers={'User-Agent'
                                 dont_filter=True)
        # 改用parse_book 作为回调函数
    def parse_book(response):
        ...
```

到此，我们介绍完了为爬虫设定起始爬取点的两种方式：

- 定义start_urls属性。
- 实现start_requests方法。

2.3.4 实现页面解析函数

页面解析函数也就是构造Request对象时通过callback参数指定的回调函数（或默认的parse方法）。页面解析函数是实现Spider中最核心的部分，它需要完成以下两项工作：

- 使用选择器提取页面中的数据，将数据封装后（Item或字典）提交给Scrapy引擎。
- 使用选择器或LinkExtractor提取页面中的链接，用其构造新的Request对象并提交给Scrapy引擎（下载链接页面）。

一个页面中可能包含多项数据以及多个链接，因此页面解析函数被要求返回一个可迭代对象（通常被实现成一个生成器函数），每次迭代返回一项数据（Item或字典）或一个Request对象。

关于如何提取数据、封装数据、提取链接等话题，我们在接下来的章节继续学习。

2.4 本章小结

本章先讲解了Scrapy的框架结构以及工作原理，然后又介绍了

Scrapy中与页面下载相关的两个核心对象Request和Response，有了这些知识的铺垫，最后我们讲解了实现一个Spider的开发流程。关于编写Spider时涉及的一些技术细节在后面章节继续讨论。

第3章 使用Selector提取数据

在第2章中，我们讲解了Spider开发的大体流程，让大家对Scrapy爬虫开发有了更加清晰的理解，接下来继续讲解编写爬虫时使用的一些具体技术，从页面中提取数据是Spider最重要的工作之一，这一章我们来学习相关内容。

3.1 Selector对象

从页面中提取数据的核心技术是HTTP文本解析，在Python中常用以下模块处理此类问题：

- BeautifulSoup

  BeautifulSoup是非常流行的HTTP解析库，API简洁易用，但解析速度较慢。

- lxml

  lxml是一套由C语言编写的xml解析库（libxml2），解析速度更快，API相对复杂。

Scrapy综合上述两者优点实现了Selector类，它是基于lxml库构建的，并简化了API接口。在Scrapy中使用Selector对象提取页面中的数据，使用时先通过XPath或CSS选择器选中页面中要提取的数据，然后进行提取。

下面详细介绍Selector对象的使用。

3.1.1 创建对象

Selector类的实现位于scrapy.selector模块，创建Selector对象时，可将页面的HTML文档字符串传递给Selector构造器方法的text参

数：

```
>>> from scrapy.selector import Selector
>>> text = '''
... <html>
...     <body>
...          <h1>Hello World</h1>
...          <h1>Hello Scrapy</h1>
...          <b>Hello python</b>
...          <ul>
...              <li>C++</li>
...              <li>Java</li>
...              <li>Python</li>
...          </ul>
...     </body>
... </html>
... '''
...
>>> selector = Selector(text=text)
<Selector xpath=None data='<html>\n       <body>\n
```

也可以使用一个Response对象构造Selector对象，将其传递给Selector构造器方法的response参数：

```
>>> from scrapy.selector import Selector
>>> from scrapy.http import HtmlResponse
>>> body = '''
... <html>
...     <body>
...          <h1>Hello World</h1>
...          <h1>Hello Scrapy</h1>
...          <b>Hello python</b>
...          <ul>
...              <li>C++</li>
...              <li>Java</li>
...              <li>Python</li>
```

```
...           </ul>
...      </body>
... </html>
... '''
...
>>> response = HtmlResponse(url='http://www.exampl
>>> selector = Selector(response=response)
>>> selector
<Selector xpath=None data='<html>\n        <body>\
```

3.1.2 选中数据

调用Selector对象的xpath方法或css方法（传入XPath或CSS选择器表达式），可以选中文档中的某个或某些部分：

```
>>> selector_list = selector.xpath('//h1')  # 选中
>>> selector_list                           # 其中
[<Selector xpath='.//h1' data='<h1>Hello World</h1
<Selector xpath='.//h1' data='<h1>Hello Scrapy</h1
```

xpath和css方法返回一个SelectorList对象，其中包含每个被选中部分对应的Selector对象，SelectorList支持列表接口，可使用for语句迭代访问其中的每一个Selector对象：

```
>>> for sel in selector_list:
...   print(sel.xpath('./text()'))
...
[<Selector xpath='./text()' data='Hello World'>]
[<Selector xpath='./text()' data='Hello Scrapy'>]
```

SelectorList对象也有xpath和css方法，调用它们的行为是：以接收到的参数分别调用其中每一个Selector对象的xpath或css方法，并将所有结果收集到一个新的SelectorList对象返回给用户。请看下面的示例：

```
>>> selector_list.xpath('./text()')
[<Selector xpath='./text()' data='Hello World'>,
 <Selector xpath='./text()' data='Hello Scrapy'>]
>>> selector.xpath('.//ul').css('li').xpath('./tex
[<Selector xpath='./text()' data='C++'>,
 <Selector xpath='./text()' data='Java'>,
 <Selector xpath='./text()' data='Python'>]
```

3.1.3　提取数据

调用Selector或SelectorLis对象的以下方法可将选中的内容提取：

- extract()
- re()
- extract_first() (SelectorList专有)
- re_first() (SelectorList专有)

首先来看extract方法，调用Selector对象的extract方法将返回选中内容的Unicode字符串：

```
>>> sl = selector.xpath('.//li')
>>> sl
[<Selector xpath='.//li' data='<li>C++</li>'>,
<Selector xpath='.//li' data='<li>Java</li>'>,
<Selector xpath='.//li' data='<li>Python</li>'>]
>>> sl[0].extract()
'<li>C++</li>'
>>> sl = selector.xpath('.//li/text()')
>>> sl
[<Selector xpath='.//li/text()' data='C++'>,
<Selector xpath='.//li/text()' data='Java'>,
<Selector xpath='.//li/text()' data='Python'>]
>>> sl[1].extract()
'Java'
```

与SelectorList对象的xpath和css方法类似，SelectorList对象的extract方法内部会调用其中每个Selector对象的extract方法，并把所有结果收集到一个列表返回给用户：

```
>>> sl = selector.xpath('.//li/text()')
>>> sl
[<Selector xpath='.//li/text()' data='C++'>,
 <Selector xpath='.//li/text()' data='Java'>,
 <Selector xpath='.//li/text()' data='Python'>]
>>> sl.extract()
['C++', 'Java', 'Python']
```

SelectorList对象还有一个extract_first方法，该方法返回其中第一个Selector对象调用extract方法的结果。通常，在SelectorList对象中只包含一个Selector对象时调用该方法，直接提取出Unicode字符串而不是列表：

```
>>> sl = selector.xpath('.//b')
>>> sl
[<Selector xpath='.//b' data='<b>Hello Python</b>'
>>> sl.extract()
['<b>Hello Python</b>']
>>> sl.extract_first()
'<b>Hello Python</b>'
```

有些时候，我们想使用正则表达式提取选中内容中的某部分，可以使用re方法（两个对象都有该方法）：

```
>>> text = '''
... <ul>
...     <li>Python 学习手册 <b>价格: 99.00 元</b></li
...     <li>Python 核心编程 <b>价格: 88.00 元</b></li
...     <li>Python 基础教程 <b>价格: 80.00 元</b></li
```

```
... </ul>
... '''
...
>>> selector = Selector(text=text)
>>> selector.xpath('.//li/b/text()')
[<Selector xpath='.//li/b/text()' data='价格: 99.0
 <Selector xpath='.//li/b/text()' data='价格: 88.0
 <Selector xpath='.//li/b/text()' data='价格: 80.0
>>> selector.xpath('.//li/b/text()').extract()
['价格: 99.00 元', '价格: 88.00 元', '价格: 80.00 元'
>>> selector.xpath('.//li/b/text()').re('\d+\.\d+'
['99.00', '88.00', '80.00']
```

SelectorList对象的re_first方法同样返回其中的第一个Selector对象调用re方法的结果：

```
>>> selector.xpath('.//li/b/text()').re_first('\d+
'99.00'
```

3.2 Response内置Selector

在实际开发中，几乎不需要手动创建Selector对象，在第一次访问一个Response对象的selector属性时，Response对象内部会以自身为参数自动创建Selector对象，并将该Selector对象缓存，以便下次使用。Scrapy源码中的相关实现如下：

```
class TextResponse(Response):

    def __init__(self, *args, **kwargs):
        ...
        self._cached_selector = None
        ...
```

```
    @property
    def selector(self):
        from scrapy.selector import Selector
        if self._cached_selector is None:
            self._cached_selector = Selector(self)
        return self._cached_selector
    ...
```

通常，我们直接使用Response对象内置的Selector对象即可：

```
>>> from scrapy.http import HtmlResponse
>>> body = '''
... <html>
...   <body>
...       <h1>Hello World</h1>
...       <h1>Hello Scrapy</h1>
...       <b>Hello python</b>
...       <ul>
...           <li>C++</li>
...           <li>Java</li>
...           <li>Python</li>
...       </ul>
...   </body>
... </html>
... '''
...
>>> response = HtmlResponse(url='http://www.exampl
>>> response.selector
<Selector xpath=None data='<html>\n        <body>\
```

为了方便用户使用，Response对象还提供了xpath和css方法，它们在内部分别调用内置Selector对象的xpath和css方法。Scrapy源码中的相关实现如下：

```
class TextResponse(Response):
```

```
    ...
    def xpath(self, query, **kwargs):
        return self.selector.xpath(query, **kwargs)

    def css(self, query):
        return self.selector.css(query)
    ...
```

使用这两个快捷方式可使代码更加简洁：

```
>>> response.xpath('.//h1/text()').extract()
['Hello World', 'Hello Scrapy']
>>> response.css('li::text').extract()
['C++', 'Java', 'Python']
```

3.3　XPath

XPath即XML路径语言（XML　Path　Language），它是一种用来确定xml文档中某部分位置的语言。

xml文档（html属于xml）是由一系列节点构成的树，例如：

```
<html>
    <body>
        <div >
            <p>Hello world<p>
            <a href="/home">Click here</a>
        </div>
    </body>
</html>
```

xml文档的节点有多种类型，其中最常用的有以下几种：

- 根节点　整个文档树的根。

- 元素节点　html、body、div、p、a。
- 属性节点　href。
- 文本节点　Hello world、Click here。

节点间的关系有以下几种：

- 父子　body是html的子节点，p和a是div的子节点。反过来，div是p和a的父节点。
- 兄弟　p和a为兄弟节点。
- 祖先／后裔　body、div、p、a都是html的后裔节点；反过来html是body、div、p、a的祖先节点。

3.3.1　基础语法

表3-1列出了XPath常用的基本语法。

表3-1　XPath常用的基本语法

表　达　式	描　　述
/	选中文档的根（root）
.	选中当前节点
..	选中当前节点的父节点
ELEMENT	选中子节点中所有 ELEMENT 元素节点
//ELEMENT	选中后代节点中所有 ELEMENT 元素节点
*	选中所有元素子节点
text()	选中所有文本子节点
@ATTR	选中名为 ATTR 的属性节点
@*	选中所有属性节点
[谓语]	谓语用来查找某个特定的节点或者包含某个特定值的节点

接下来，我们通过一些例子展示XPath的使用。

首先创建一个用于演示的html文档，并用其构造一个HtmlResponse对象：

```
>>> from scrapy.selector import Selector
>>> from scrapy.http import HtmlResponse
>>> body = '''
```

```
... <html>
...     <head>
...          <base href='http://example.com/' />
...          <title>Example website</title>
...     </head>
...     <body>
...          <div id='images'>
...              <a href='image1.html'>Name: Image
...              <a href='image2.html'>Name: Image
...              <a href='image3.html'>Name: Image
...              <a href='image4.html'>Name: Image
...              <a href='image5.html'>Name: Image
...          </div>
...     </body>
... </html>
... '''
...
>>> response = HtmlResponse(url='http://www.exampl
```

- /：描述一个从根开始的绝对路径。

```
>>> response.xpath('/html')
[<Selector xpath='/html' data='<html>\n\t<head>\n\
>>> response.xpath('/html/head')
[<Selector xpath='/html/head' data='<head>\n\t\t<b
```

- E1/E2：选中E1子节点中的所有E2。

```
# 选中div子节点中的所有a
>>> response.xpath('/html/body/div/a')
[<Selector xpath='/html/body/div/a' data='<a href=
<Selector xpath='/html/body/div/a' data='<a href="
<Selector xpath='/html/body/div/a' data='<a href="
<Selector xpath='/html/body/div/a' data='<a href="
<Selector xpath='/html/body/div/a' data='<a href="
```

- //E：选中文档中的所有E，无论在什么位置。

```
# 选中文档中的所有a
>>> response.xpath('//a')
[<Selector xpath='//a' data='<a href="image1.html"
<Selector xpath='//a' data='<a href="image2.html">
<Selector xpath='//a' data='<a href="image3.html">
<Selector xpath='//a' data='<a href="image4.html">
<Selector xpath='//a' data='<a href="image5.html">
```

- E1//E2：选中E1后代节点中的所有E2，无论在后代中的什么位置。

```
# 选中body后代中的所有img
>>> response.xpath('/html/body//img')
[<Selector xpath='/html/body//img' data='<img src=
 <Selector xpath='/html/body//img' data='<img src=
 <Selector xpath='/html/body//img' data='<img src=
 <Selector xpath='/html/body//img' data='<img src=
 <Selector xpath='/html/body//img' data='<img src=
```

- E/text()：选中E的文本子节点。

```
# 选中所有a的文本
>>> sel = response.xpath('//a/text()')
>>> sel
[<Selector xpath='//a/text()' data='Name: My image
<Selector xpath='//a/text()' data='Name: My image
<Selector xpath='//a/text()' data='Name: My image
<Selector xpath='//a/text()' data='Name: My image
<Selector xpath='//a/text()' data='Name: My image
>>> sel.extract()
['Name: My image 1 ',
'Name: My image 2 ',
'Name: My image 3 ',
```

```
'Name: My image 4 ',
'Name: My image 5 ']
```

- E/*：选中E的所有元素子节点。

```
# 选中html的所有元素子节点
>>> response.xpath('/html/*')
[<Selector xpath='/html/*' data='<head>\n\t\t<base
 <Selector xpath='/html/*' data='<body>\n\t\t<div

# 选中div的所有后代元素节点
>>> response.xpath('/html/body/div//*')
[<Selector xpath='/html/body/div//*' data='<a href
 <Selector xpath='/html/body/div//*' data='<br>'>,
 <Selector xpath='/html/body/div//*' data='<img sr
 <Selector xpath='/html/body/div//*' data='<a href
 <Selector xpath='/html/body/div//*' data='<br>'>,
 <Selector xpath='/html/body/div//*' data='<img sr
 <Selector xpath='/html/body/div//*' data='<a href
 <Selector xpath='/html/body/div//*' data='<br>'>,
 <Selector xpath='/html/body/div//*' data='<img sr
 <Selector xpath='/html/body/div//*' data='<a href
 <Selector xpath='/html/body/div//*' data='<br>'>,
 <Selector xpath='/html/body/div//*' data='<img sr
 <Selector xpath='/html/body/div//*' data='<a href
 <Selector xpath='/html/body/div//*' data='<br>'>,
 <Selector xpath='/html/body/div//*' data='<img sr
```

- */E：选中孙节点中的所有E。

```
# 选中div孙节点中的所有img
>>> response.xpath('//div/*/img')
[<Selector xpath='//div/*/img' data='<img src="ima
 <Selector xpath='//div/*/img' data='<img src="ima
 <Selector xpath='//div/*/img' data='<img src="ima
```

```
<Selector xpath='//div/*/img' data='<img src="ima
<Selector xpath='//div/*/img' data='<img src="ima
```

- E/@ATTR：选中E的ATTR属性。

```
# 选中所有img的src 属性
>>> response.xpath('//img/@src')
[<Selector xpath='//img/@src' data='image1.jpg'>,
<Selector xpath='//img/@src' data='image2.jpg'>,
<Selector xpath='//img/@src' data='image3.jpg'>,
<Selector xpath='//img/@src' data='image4.jpg'>,
<Selector xpath='//img/@src' data='image5.jpg'>]
```

- //@ATTR：选中文档中所有ATTR属性。

```
# 选中所有的href 属性
>>> response.xpath('//@href')
[<Selector xpath='//@href' data='http://example.cc
<Selector xpath='//@href' data='image1.html'>,
<Selector xpath='//@href' data='image2.html'>,
<Selector xpath='//@href' data='image3.html'>,
<Selector xpath='//@href' data='image4.html'>,
<Selector xpath='//@href' data='image5.html'>]
```

- E/@*：选中E的所有属性。

```
# 获取第一个a 下img的所有属性（这里只有src 一个属性）
>>> response.xpath('//a[1]/img/@*')
[<Selector xpath='//a[1]/img/@*' data='image1.jpg'
```

- .：选中当前节点，用来描述相对路径。

```
# 获取第1个a的选择器对象
```

```
>>> sel = response.xpath('//a')[0]
>>> sel
<Selector xpath='//a' data='<a href="image1.html">

# 假设我们想选中当前这个a 后代中的所有img，下面的做法是错误
# 会找到文档中所有的img
# 因为//img是绝对路径，会从文档的根开始搜索，而不是从当前的
>>> sel.xpath('//img')
[<Selector xpath='//img' data='<img src="image1.jp
<Selector xpath='//img' data='<img src="image2.jpg
<Selector xpath='//img' data='<img src="image3.jpg
<Selector xpath='//img' data='<img src="image4.jpg
<Selector xpath='//img' data='<img src="image5.jpg
# 需要使用.//img 来描述当前节点后代中的所有img
>>> sel.xpath('.//img')
[<Selector xpath='.//img' data='<img src="image1.j
```

- ..：选中当前节点的父节点，用来描述相对路径。

```
# 选中所有img的父节点
>>> response.xpath('//img/..')
[<Selector xpath='//img/..' data='<a href="image1.
 <Selector xpath='//img/..' data='<a href="image2.
 <Selector xpath='//img/..' data='<a href="image3.
 <Selector xpath='//img/..' data='<a href="image4.
 <Selector xpath='//img/..' data='<a href="image5.
```

- node[谓语]：谓语用来查找某个特定的节点或者包含某个特定值的节点。

```
# 选中所有a 中的第3 个
>>> response.xpath('//a[3]')
[<Selector xpath='//a[3]' data='<a href="image3.ht
```

```
# 使用last函数，选中最后1 个
>>> response.xpath('//a[last()]')
[<Selector xpath='//a[last()]' data='<a href="imag

# 使用position函数，选中前3 个
>>> response.xpath('//a[position()<=3]')
[<Selector xpath='//a[position()<=3]' data='<a hre
 <Selector xpath='//a[position()<=3]' data='<a hre
 <Selector xpath='//a[position()<=3]' data='<a hre

# 选中所有含有id属性的div
>>> response.xpath('//div[@id]')
[<Selector xpath='//div[@id]' data='<div id="image

# 选中所有含有id属性且值为"images"的div
>>> response.xpath('//div[@id="images"]')
[<Selector xpath='//div[@id="images"]' data='<div
```

3.3.2 常用函数

XPath还提供许多函数，如数字、字符串、时间、日期、统计等。在上面的例子中，我们已经使用了函数position()、last()。由于篇幅有限，下面仅介绍两个十分常用的字符串函数。

- string(arg)：返回参数的字符串值。

```
>>> from scrapy.selector import Selector
>>> text='<a href="#">Click here to go to the <str
>>> sel = Selector(text=text)
>>> sel
<Selector xpath=None data='<html><body><a href="#"
# 以下做法和sel.xpath('/html/body/a/strong/text()')
>>> sel.xpath('string(/html/body/a/strong)').extra
['Next Page']
```

```
# 如果想得到a 中的整个字符串'Click here to go to the N
# 使用text()就不行了，因为Click here to go to the和Ne
# 以下做法将得到两个子串
>>> sel.xpath('/html/body/a//text()').extract()
['Click here to go to the ', 'Next Page']
# 这种情况下可以使用string()函数
>>> sel.xpath('string(/html/body/a)').extract()
['Click here to go to the Next Page']
```

- contains(str1,　str2)：判断str1中是否包含str2，返回布尔值。

```
>>> text = '''
... <div>
...    <p class="small info">hello world</p>
...    <p class="normal info">hello scrapy</p>
... </div>
... '''
>>> sel = Selector(text=text)
>>> sel.xpath('//p[contains(@class, "small")]') #
[<Selector xpath='//p[contains(@class, "small")]'
>>> sel.xpath('//p[contains(@class, "info")]') # 选
[<Selector xpath='//p[contains(@class, "info")]' d
 <Selector xpath='//p[contains(@class, "info")]' d
```

关于XPath的使用先介绍到这里，更多详细内容可以参看XPath文档：https://www.w3.org/TR/xpath/。

3.4　CSS选择器

CSS即层叠样式表，其选择器是一种用来确定HTML文档中某部分位置的语言。

CSS选择器的语法比XPath更简单一些，但功能不如XPath强大。实际上，当我们调用Selector对象的CSS方法时，在其内部会使用Python库cssselect将CSS选择器表达式翻译成XPath表达式，然后调用

Selector对象的XPATH方法。

表3-2列出了CSS选择器的一些基本语法。

表3-2　CSS选择器

表 达 式	描　　述	例　　子
*	选中所有元素	*
E	选中 E 元素	p
E1,E2	选中 E1 和 E2 元素	div,pre
E1 E2	选中 E1 后代元素中的 E2 元素	div p
E1>E2	选中 E1 子元素中的 E2 元素	div>p
E1+E2	选中 E1 兄弟元素中的 E2 元素	p+strong
.CLASS	选中 CLASS 属性包含 CLASS 的元素	.info
#ID	选中 id 属性为 ID 的元素	#main
[ATTR]	选中包含 ATTR 属性的元素	[href]
[ATTR=VALUE]	选中包含 ATTR 属性且值为 VALUE 的元素	[method=post]
[ATTR~=VALUE]	选中包含 ATTR 属性且包含 VALUE 的元素	[class~=clearfix]
E:nth-child(n) E:nth-last-child(n)	选中 E 元素，且该元素必须是其父元素的（倒数）第 n 个子元素	a:nth-child(1) a:nth-last-child(2)
E:first-child E:last-child	选中 E 元素，且该元素必须是其父元素的（倒数）第一个子元素	a:first-child a:last-child
E:empty	选中没有子元素的 E 元素	div:empty
E::text	选中 E 元素的文本节点（Text Node）	p::text

和学习XPath一样，通过一些例子展示CSS选择器的使用。

先创建一个HTML文档并构造一个HtmlResponse对象：

```
>>> from scrapy.selector import Selector
>>> from scrapy.http import HtmlResponse
>>> body = '''
... <html>
...     <head>
...         <base href='http://example.com/' />
...         <title>Example website</title>
...     </head>
...     <body>
...         <div id='images-1' style="width: 1230p
...             <a href='image1.html'>Name: Image
...             <a href='image2.html'>Name: Image
...             <a href='image3.html'>Name: Image
...         </div>
```

```
...
...            <div id='images-2' class='small'>
...                <a href='image4.html'>Name: Image
...                <a href='image5.html'>Name: Image
...            </div>
...       </body>
... </html>
... '''
...
>>> response = HtmlResponse(url='http://www.exampl
```

- E：选中E元素。

```
# 选中所有的img
>>> response.css('img')
[<Selector xpath='descendant-or-self::img' data='<
 <Selector xpath='descendant-or-self::img' data='<
 <Selector xpath='descendant-or-self::img' data='<
 <Selector xpath='descendant-or-self::img' data='<
 <Selector xpath='descendant-or-self::img' data='<
```

- E1,E2：选中E1和E2元素。

```
    # 选中所有base和title
    >>> response.css('base,title')
    [<Selector xpath='descendant-or-self::base | d
href="http://example.com/">'>,
     <Selector xpath='descendant-or-self::base | d
website</title>'>]
```

- E1 E2：选中E1后代元素中的E2元素。

```
# div 后代中的img
>>> response.css('div img')
[<Selector xpath='descendant-or-self::div/descenda
```

```
 <Selector xpath='descendant-or-self::div/descenda
 <Selector xpath='descendant-or-self::div/descenda
 <Selector xpath='descendant-or-self::div/descenda
 <Selector xpath='descendant-or-self::div/descenda
```

- E1＞E2：选中E1子元素中的E2元素。

```
# body 子元素中的div
>>> response.css('body>div')
[<Selector xpath='descendant-or-self::body/div' da
 <Selector xpath='descendant-or-self::body/div' da
```

- [ATTR]：选中包含ATTR属性的元素。

```
# 选中包含style属性的元素
>>> response.css('[style]')
[<Selector xpath='descendant-or-self::*[@style]' d
```

- [ATTR＝VALUE]：选中包含ATTR属性且值为VALUE的元素。

```
     # 选中属性id值为images-1的元素
     >>> response.css('[id=images-1]')
     [<Selector xpath="descendant-or-self::*[@id =
1230px;'>]
```

- E:nth-child(n)：选中E元素，且该元素必须是其父元素的第n个子元素。

```
    # 选中每个div的第一个a
    >>> response.css('div>a:nth-child(1)')
    [<Selector xpath="descendant-or-self::div/*[na
```

```
href="image1.html">Name: Image 1 <br>'>,
     <Selector xpath="descendant-or-self::div/*[na
href="image4.html">Name: Image 4 <br>'>]


    # 选中第二个div的第一个a
    >>> response.css('div:nth-child(2)>a:nth-child
    [<Selector xpath="descendant-or-self::*/*[name
(position() = 1)]" data='<a href="image4.html">Nam
```

- E:first-child：选中E元素，该元素必须是其父元素的第一个子元素。
- E:last-child：选中E元素，该元素必须是其父元素的倒数第一个子元素。

```
     # 选中第一个div的最后一个a
     >>> response.css('div:first-child>a:last-chil
     [<Selector xpath="descendant-or-self::*/*[nam
(position() = last())]" data='<a href="image3.html
```

- E::text：选中E元素的文本节点。

```
# 选中所有a的文本
>>> sel = response.css('a::text')
>>> sel
[<Selector xpath='descendant-or-self::a/text()' da
 <Selector xpath='descendant-or-self::a/text()' da
 <Selector xpath='descendant-or-self::a/text()' da
 <Selector xpath='descendant-or-self::a/text()' da
 <Selector xpath='descendant-or-self::a/text()' da
>>> sel.extract()
['Name: Image 1 ',
 'Name: Image 2 ',
 'Name: Image 3 ',
 'Name: Image 4 ',
```

```
    'Name: Image 5 ']
```

关于CSS选择器的使用先介绍到这里，更多详细内容可以参看CSS选择器文档：https://www.w3.org/TR/css3-selectors/。

3.5　本章小结

本章学习了从页面中提取数据的相关内容，首先带大家了解了Scrapy中的Selector对象，然后学习如何使用Selector对象在页面中选中并提取数据，最后通过一系列例子讲解了XPath和CSS选择器的用法。

第4章 使用Item封装数据

在第3章中，我们学习了从页面中提取数据的方法，本章来学习如何封装爬取到的数据。以爬取某图书网站的书籍信息为例，对于网站中的每一本书可以提取出书名、价格、作者、出版社、出版时间等多个信息字段。应该用怎样的数据结构来维护这些零散的信息字段呢？最容易想到是使用Python字典（dict）。

回顾第1章example项目中BooksSpider的代码：

```
class BooksSpider(scrapy.Spider):
    ...
    def parse(self, response):
        for sel in response.css('article.product_po
            name = sel.xpath('./h3/a/@title').extra
            price = sel.css('p.price_color::text').
            yield {
                'name': name,
                'price': price,
            }
    ...
```

在该案例中，我们便使用了Python字典存储一本书的信息，但字典可能有以下缺点：

（1）无法一目了然地了解数据中包含哪些字段，影响代码可读性。

（2）缺乏对字段名字的检测，容易因程序员的笔误而出错。

（3）不便于携带元数据（传递给其他组件的信息）。

为解决上述问题，在Scrapy中可以使用自定义的Item类封装爬取到的数据。

4.1 Item和Field

Scrapy提供了以下两个类，用户可以使用它们自定义数据类（如书籍信息），封装爬取到的数据：

- Item基类

  自定义数据类（如BookItem）的基类。

- Field类

  用来描述自定义数据类包含哪些字段（如name、price等）。

自定义一个数据类，只需继承Item，并创建一系列Field对象的类属性（类似于在Django中自定义Model）即可。以定义书籍信息BookItem为例，它包含两个字段，分别为书的名字name和书的价格price，代码如下：

```
>>> from scrapy import Item, Field
>>> class BookItem(Item):
...     name = Field()
...     price = Field()
```

Item支持字典接口，因此BookItem在使用上和Python字典类似，可按以下方式创建BookItem对象：

```
>>> book1 = BookItem(name='Needful Things', price=
>>> book1
{'name': 'Needful Things', 'price': 45.0}
>>> book2 = BookItem()
>>> book2
{}
>>> book2['name'] = 'Life of Pi'
>>> book2['price'] = 32.5
{'name': 'Life of Pi', 'price': 32.5}
```

对字段进行赋值时，BookItem内部会对字段名进行检测，如果赋值一个没有定义的字段，就会抛出异常（防止因用户粗心而导致错误）：

```
>>> book = BookItem()
>>> book['name'] = 'Memoirs of a Geisha'
>>> book['prize'] = 43.0 # 粗心，把price 拼写成了pri
Traceback (most recent call last):
   ...
KeyError: 'BookItem does not support field: prize'
```

访问BookItem对象中的字段与访问字典类似，示例如下：

```
>>> book = BookItem(name='Needful Things', price=4
>>> book['name']
'Needful Things'
>>> book.get('price', 60.0)
45.0
>>> list(book.items())
[('price', 45.0), ('name', 'Needful Things')]
```

接下来，我们改写第1章example项目中的代码，使用Item和Field定义BookItem类，用其封装爬取到的书籍信息项目目录下的items.py文件供用户实现各种自定义的数据类，在items.py中实现BookItem，代码如下：

```
from scrapy import Item, Field


class BookItem(Item):
    name = Field()
    price = Field()
```

修改之前的BooksSpider，使用BookItem替代Python字典，代码如下：

```
from ..items import BookItem


class BooksSpider(scrapy.Spider):
   ...
   def parse(self, response):
      for sel in response.css('article.product_pod
         book = BookItem()
       book['name'] = sel.xpath('./h3/a/@title').ext
       book['price'] = sel.css('p.price_color::text'
       yield book
...
```

4.2　拓展Item子类

有些时候，我们可能要根据需求对已有的自定义数据类（Item子类）进行拓展。例如，example项目中又添加了一个新的Spider，它负责在另外的图书网站爬取国外书籍（中文翻译版）的信息，此类书籍的信息比之前多了一个译者字段，此时可以继承BookItem定义一个ForeignBookItem类，在其中添加一个译者字段，代码如下：

```
>>> class ForeignBookItem(BookItem):
...   translator = Field()
...
>>> book = ForeignBookItem()
>>> book['name'] = '巴黎圣母院'
>>> book['price'] = 20.0
>>> book['translator'] = '陈敬容'
```

4.3　Field元数据

在第2章中曾讲到，一项数据由Spider提交给Scrapy引擎后，可能会被递送给其他组件（Item Pipeline、Exporter）处理。假设想传递额外信息给处理数据的某个组件（例如，告诉该组件应以怎样的方式处理数据），此时可以使用Field的元数据。请看下面的例子：

```
class ExampleItem(Item):
    x = Field(a='hello', b=[1, 2, 3])          # x 有两
    y = Field(a=lambda x: x ** 2)              # y 有一
```

访问一个ExampleItem对象的fields属性，将得到一个包含所有Field对象的字典：

```
>>> e = ExampleItem(x=100, y=200)
>>> e.fields
{'x': {'a': 'hello', 'b': [1, 2, 3]},
 'y': {'a': <function __main__.ExampleItem.<lambda
>>> type(e.fields['x'])
scrapy.item.Field
>>> type(e.fields['y'])
scrapy.item.Field
```

实际上，Field是Python字典的子类，可以通过键获取Field对象中的元数据：

```
>>> issubclass(Field, dict)
True
>>> field_x = e.fields['x']       # 注意，不要混淆e.f
>>> field_x
{'a': 'hello', 'b': [1, 2, 3]}
>>> field_x['a']
'hello'
>>> field_y = e.fields['y']
>>> field_y
{'a': <function __main__.ExampleItem.<lambda>>}
```

```
>>> field_y.get('a', lambda x: x)
<function __main__.ExampleItem.<lambda>>
```

接下来，看一个应用Field元数据的实际例子。假设我们要把爬取到的书籍信息写入csv文件，那每一项数据最终由Scrapy提供的CsvItemExporter写入文件（数据导出在第7章详细讲解），在爬取过程中提取到的信息并不总是一个字符串，有时可能是一个字符串列表，例如：

```
>>> book['authors'] = ['李雷', '韩梅梅', '吉姆']
```

但在写入csv文件时，需要将列表内所有字符串串行化成一个字符串，串行化的方式有很多种，例如：

```
1. '李雷|韩梅梅|吉姆'                # '|'.join(book
2. '李雷;韩梅梅;吉姆'                # ';'.join(book
3. "['李雷', '韩梅梅', '吉姆']" # str(book['authors'
```

我们可以通过authors字段的元数据告诉CsvItemExporter如何对authors字段串行化：

```
class BookItem(Item):
    ...
    authors = Field(serializer=lambda x: '|'.join(x
    ...
```

其中，元数据的键serializer是CsvItemExporter规定好的，它会用该键获取元数据，即一个串行化函数对象，并使用这个串行化函数将authors字段串行化成一个字符串。以下是Scrapy源码中的相关实现：

```
# exports.py


class BaseItemExporter(object):
    ...


    def _get_serialized_fields(self, item, default_
        ...
        for field_name in field_iter:
            if field_name in item:
                field = {} if isinstance(item, dict)
                value = self.serialize_field(field,
            else:
                value = default_value
            yield field_name, value
    ...


class CsvItemExporter(BaseItemExporter):



    ...


    def export_item(self, item):
        ...
        fields = self._get_serialized_fields(item,
                                             include_em
        values = list(self._build_row(x for _, x in
        self.csv_writer.writerow(values)



    ...



    def serialize_field(self, field, name, value):
        serializer = field.get('serializer', self._
        return serializer(value)
    ...
```

解释上述代码如下：

- 爬取到的每一项数据由export_item方法导出到文件，写入文件之前，先调用_get_serialized_fields方法（在基类中实现）获得数据中每个字段串行化的结果。
- 在_get_serialized_fields方法中调用serialize_field方法，获取其中一个字段串行化的结果。
- 在serialize_field方法中获取字段的元数据serializer，得到串行化函数（如果不存在，就使用默认的_join_if_needed函数），最终调用该函数对字段串行化，并将结果返回。

在实际应用中，我们可以仿照上面的例子灵活使用Field元数据。

4.4 本章小结

本章介绍了在Scrapy中如何封装爬取到的数据，先了解了Item基类以及用来定义字段的Field类，然后展示了一个使用它们封装数据的例子。最后，还介绍了使用Field元数据给其他组件传递信息的方法。

第5章 使用Item Pipeline处理数据

在之前的章节中，我们学习了提取数据以及封装数据的方法，这一章来学习如何对爬取到的数据进行处理。在Scrapy中，Item Pipeline是处理数据的组件，一个Item Pipeline就是一个包含特定接口的类，通常只负责一种功能的数据处理，在一个项目中可以同时启用多个Item Pipeline，它们按指定次序级联起来，形成一条数据处理流水线。

以下是Item Pipeline的几种典型应用：

- 清洗数据。
- 验证数据的有效性。
- 过滤掉重复的数据。
- 将数据存入数据库。

5.1 Item Pipeline

通过一个例子讲解Item Pipeline的使用，在第1章的example项目中，我们爬取到的书籍价格是以英镑为单位的：

```
$ scrapy crawl books -o books.csv
...
$ head -5 books.csv     # 查看文件开头的5 行
name,price
A Light in the Attic,£51.77
Tipping the Velvet,£53.74
Soumission,£50.10
Sharp Objects,£47.82
```

如果我们期望爬取到的书价是人民币价格，就需要用英镑价格乘以汇率计算出人民币价格（处理数据），此时可以实现一个价格转换的Item Pipeline来完成这个工作。接下来在example项目中实现它。

5.1.1 实现Item Pipeline

在创建一个Scrapy项目时，会自动生成一个pipelines.py文件，它用来放置用户自定义的Item Pipeline，在example项目的pipelines.py中实现PriceConverterPipeline，代码如下：

```
class PriceConverterPipeline(object):

    # 英镑兑换人民币汇率
    exchange_rate = 8.5309

    def process_item(self, item, spider):
        # 提取item的price 字段（如£53.74）
        # 去掉前面英镑符号£，转换为float 类型，乘以汇率
        price = float(item['price'][1:]) * self.exc

        # 保留2 位小数，赋值回item的price 字段
        item['price'] = '￥%.2f' % price

        return item
```

对上述代码解释如下：

- 一个Item Pipeline不需要继承特定基类，只需要实现某些特定方法，例如process_item、open_spider、close_spider。
- 一个Item Pipeline必须实现一个process_item(item, spider)方法，该方法用来处理每一项由Spider爬取到的数据，其中的

两个参数：

- Item　爬取到的一项数据（Item或字典）。
- Spider　爬取此项数据的Spider对象。

上述代码中的process_item方法实现非常简单，将书籍的英镑价格转换为浮点数，乘以汇率并保留2位小数，然后赋值回item的price字段，最后返回被处理过的item。

可以看出，process_item方法是Item　Pipeline的核心，对该方法还需再做两点补充说明：

- 如果process_item在处理某项item时返回了一项数据（Item或字典），返回的数据会递送给下一级Item　Pipeline（如果有）继续处理。
- 如果process_item在处理某项item时抛出（raise）一个DropItem异常（scrapy.exceptions.DropItem），该项item便会被抛弃，不再递送给后面的Item　Pipeline继续处理，也不会导出到文件。通常，我们在检测到无效数据或想要过滤数据时，抛出DropItem异常。

除了必须实现的process_item方法外，还有3个比较常用的方法，可根据需求选择实现：

- open_spider(self, spider)

  Spider打开时（处理数据前）回调该方法，通常该方法用于在开始处理数据之前完成某些初始化工作，如连接数据库。

- close_spider(self, spider)

  Spider关闭时（处理数据后）回调该方法，通常该方法用于在处理完所有数据之后完成某些清理工作，如关闭数据库。

- from_crawler(cls, crawler)

  创建Item　Pipeline对象时回调该类方法。通常，在该方法中通过crawler.settings读取配置，根据配置创建Item　Pipeline对象。

在后面的例子中，我们展示了以上方法的应用场景。

5.1.2　启用Item Pipeline

在Scrapy中，Item　Pipeline是可选的组件，想要启用某个（或某些）Item Pipeline，需要在配置文件settings.py中进行配置：

```
ITEM_PIPELINES = {
  'example.pipelines.PriceConverterPipeline': 300,
}
```

ITEM_PIPELINES是一个字典，我们把想要启用的Item　Pipeline添加到这个字典中，其中每一项的键是每一个Item　Pipeline类的导入路径，值是一个0～1000的数字，同时启用多个Item　Pipeline时，Scrapy根据这些数值决定各Item　Pipeline处理数据的先后次序，数值小的在前。

启用PriceConverterPipeline后，重新运行爬虫，并观察结果：

```
$ scrapy crawl books -o books.csv
...
$ head -5 books.csv # 查看文件开头的5 行
name,price
A Light in the Attic,￥441.64
Tipping the Velvet,￥458.45
Soumission,￥427.40
Sharp Objects,￥407.95
```

使用PriceConverterPipeline对数据进行处理后，books.csv中的书价转换成了人民币价格。

5.2　更多例子

我们通过一个例子学习了如何使用Item Pipeline对数据进行处理，下面再看两个实际例子。

5.2.1 过滤重复数据

为了确保爬取到的书籍信息中没有重复项，可以实现一个去重Item Pipeline。这里，我们就以书名作为主键（实际应以ISBN编号为主键，但是仅爬取了书名和价格）进行去重，实现DuplicatesPipeline代码如下：

```
from scrapy.exceptions import DropItem

class DuplicatesPipeline(object):

   def __init__(self):
   self.book_set = set()

def process_item(self, item, spider):
   name = item['name']
   if name in self.book_set:
       raise DropItem("Duplicate book found: %s" %

   self.book_set.add(name)
   return item
```

对上述代码解释如下：

- 增加构造器方法，在其中初始化用于对书名去重的集合。
- 在process_item方法中，先取出item的name字段，检查书名是否已在集合book_set中，如果存在，就是重复数据，抛出DropItem异常，将item抛弃；否则，将item的name字段存

入集合，返回item。

接下来测试DuplicatesPipeline。首先在不启用DuplicatesPipeline的情况下，运行爬虫，查看结果：

```
$ scrapy crawl books -o book1.csv
...
$ cat -n book1.csv
     1 price,name
     2 ￥441.64,A Light in the Attic
     3 ￥458.45,Tipping the Velvet
     4 ￥427.40,Soumission
     5 ￥407.95,Sharp Objects
     6 ￥462.63,Sapiens: A Brief History of Humanki
     7 ￥193.22,The Requiem Red
     8 ￥284.42,The Dirty Little Secrets of Getting
    ...
   993 ￥317.86,Bounty (Colorado Mountain #7)
   994 ￥173.18,Blood Defense (Samantha Brinkman #1
   995 ￥295.60,"Bleach, Vol. 1: Strawberry and the
   996 ￥370.07,Beyond Good and Evil
   997 ￥473.72,Alice in Wonderland (Alice's Advent
   998 ￥486.77,"Ajin: Demi-Human, Volume 1 (Ajin: 
   999 ￥144.77,A Spy's Devotion (The Regency Spies
  1000 ￥460.50,1st to Die (Women's Murder Club #1)
  1001 ￥222.49,"1,000 Places to See Before You Die
```

此时有1000本书。

然后在配置文件settings.py中启用DuplicatesPipeline：

```
ITEM_PIPELINES = {
   'example.pipelines.PriceConverterPipeline': 300
   'example.pipelines.DuplicatesPipeline': 350,
}
```

运行爬虫，对比结果：

```
$ scrapy crawl books -o book2.csv
...
$ cat -n book2.csv
     1   name,price
     2   A Light in the Attic,¥441.64
     3   Tipping the Velvet,¥458.45
     4   Soumission,¥427.40
     5   Sharp Objects,¥407.95
     6   Sapiens: A Brief History of Humankind,¥46
     7   The Requiem Red,¥193.22
     8   The Dirty Little Secrets of Getting Your D
   ...
   993   Blood Defense (Samantha Brinkman #1),¥173
   994   "Bleach, Vol. 1: Strawberry and the Soul R
   995   Beyond Good and Evil,¥370.07
   996   Alice in Wonderland (Alice's Adventures in
   997   "Ajin: Demi-Human, Volume 1 (Ajin: Demi-Hu
   998   A Spy's Devotion (The Regency Spies of Lon
   999   1st to Die (Women's Murder Club #1),¥460.
  1000   "1,000 Places to See Before You Die",¥222
```

只有999本了，比之前少了1本，说明有两本书是同名的，翻阅爬虫的log信息可以找到重复项：

```
[scrapy.core.scraper] WARNING: Dropped: Duplicate
{'name': 'The Star-Touched Queen', 'price': '¥275
```

5.2.2　将数据存入MongoDB

有时，我们想把爬取到的数据存入某种数据库中，可以实现Item Pipeline完成此类任务。下面实现一个能将数据存入MongoDB数据库的Item Pipeline，代码如下：

```
from scrapy.item import Item
```

```
import pymongo

class MongoDBPipeline(object):

   DB_URI = 'mongodb://localhost:27017/'
   DB_NAME = 'scrapy_data'

   def open_spider(self, spider):
      self.client = pymongo.MongoClient(self.DB_UR
      self.db = self.client[self.DB_NAME]

   def close_spider(self, spider):
      self.client.close()

   def process_item(self, item, spider):
      collection = self.db[spider.name]
      post = dict(item) if isinstance(item, Item)
      collection.insert_one(post)
      return item
```

对上述代码解释如下。

- 在类属性中定义两个常量：
  - DB_URI　数据库的URI地址。
  - DB_NAME　数据库的名字。
- 在Spider整个爬取过程中，数据库的连接和关闭操作只需要进行一次，应在开始处理数据之前连接数据库，并在处理完所有数据之后关闭数据库。因此实现以下两个方法（在Spider打开和关闭时被调用）：
  - open_spider(spider)
  - close_spider(spider)

分别在open_spider和close_spider方法中实现数据库的连接与关闭。

- 在process_item中实现MongoDB数据库的写入操作，使用self.db和spider.name获取一个集合（collection），然后将数据插入该集合，集合对象的insert_one方法需传入一个字典对象（不能传入Item对象），因此在调用前先对item的类型进行判断，如果item是Item对象，就将其转换为字典。

接下来测试MongoDBPipeline，在配置文件settings.py中启用MongoDBPipeline：

```
ITEM_PIPELINES = {
   'example.pipelines.PriceConverterPipeline': 300
   'example.pipelines.MongoDBPipeline': 400,
}
```

运行爬虫，并查看数据库中的结果：

```
     $ scrapy crawl books
     ...
     $ mongo
     MongoDB shell version: 2.4.9
     connecting to: test
     > use scrapy_data
     switched to db scrapy_data
     > db.books.count()
     1000
     > db.books.find()
     { "_id" : ObjectId("58ae39a89dcd191973cc588f"
Attic" }
     { "_id" : ObjectId("58ae39a89dcd191973cc5890"
Velvet" }
     { "_id" : ObjectId("58ae39a89dcd191973cc5891"
     { "_id" : ObjectId("58ae39a89dcd191973cc5892"
     { "_id" : ObjectId("58ae39a89dcd191973cc5893"
```

```
History of Humankind" }
     { "_id" : ObjectId("58ae39a89dcd191973cc5894"
Red" }
     { "_id" : ObjectId("58ae39a89dcd191973cc5895"
Secrets of Getting Your Dream Job" }
     { "_id" : ObjectId("58ae39a89dcd191973cc5896"
Woman: A Novel Based on the Life of the Infamous F
     { "_id" : ObjectId("58ae39a89dcd191973cc5897"
Boat: Nine Americans and Their Epic Quest for Gold
     { "_id" : ObjectId("58ae39a89dcd191973cc5898"
     { "_id" : ObjectId("58ae39a89dcd191973cc5899"
(Triangular Trade Trilogy, #1)" }
     { "_id" : ObjectId("58ae39a89dcd191973cc589a"
Sonnets" }
     { "_id" : ObjectId("58ae39a89dcd191973cc589b"
     { "_id" : ObjectId("58ae39a89dcd191973cc589c"
Precious Little Life (Scott Pilgrim #1)" }
     { "_id" : ObjectId("58ae39a89dcd191973cc589d"
Again" }
     { "_id" : ObjectId("58ae39a89dcd191973cc589e"
Your Life: Scenes from the American Indie Undergro
     { "_id" : ObjectId("58ae39a89dcd191973cc589f"
     { "_id" : ObjectId("58ae39a89dcd191973cc58a0"
Best Science Fiction Stories 1800-1849" }
     { "_id" : ObjectId("58ae39a89dcd191973cc58a1"
Beginners" }
     { "_id" : ObjectId("58ae39a89dcd191973cc58a2"
Himalayas" }
     Type "it" for more
```

在上述实现中，数据库的URI地址和数据库的名字硬编码在代码中，如果希望通过配置文件设置它们，只需稍作改动，代码如下：

```
from scrapy.item import Item
import pymongo


class MongoDBPipeline(object):
```

```
    @classmethod
    def from_crawler(cls, crawler):
        cls.DB_URI = crawler.settings.get('MONGO_DB_
                                          'mongodb://loca
        cls.DB_NAME = crawler.settings.get('MONGO_DB

        return cls()

    def open_spider(self, spider):
        self.client = pymongo.MongoClient(self.DB_UR
        self.db = self.client[self.DB_NAME]
def close_spider(self, spider):
    self.client.close()

def process_item(self, item, spider):
    collection = self.db[spider.name]
    post = dict(item) if isinstance(item, Item) els
    collection.insert_one(post)

    return item
```

对上述改动解释如下：

- 增加类方法from_crawler（cls, crawler），替代在类属性中定义DB_URI和DB_NAME。
- 如果一个Item Pipeline定义了from_crawler方法，Scrapy就会调用该方法来创建Item Pipeline对象。该方法有两个参数：
  - cls Item Pipeline类的对象（这里为MongoDBPipeline类对象）。
  - crawler Crawler是Scrapy中的一个核心对象，可以通过crawler的settings属性访问配置文件。

- 在from_crawler方法中，读取配置文件中的MONGO_DB_URI和MONGO_DB_NAME（不存在使用默认值），赋给cls的属性，即MongoDBPipeline类属性。
- 其他代码并没有任何改变，因为这里只是改变了设置MongoDBPipeline类属性的方式。

现在，我们可在配置文件settings.py中对所要使用的数据库进行设置：

```
MONGO_DB_URI = 'mongodb://192.168.1.105:27017/'
MONGO_DB_NAME = 'liushuo_scrapy_data'
```

5.3　本章小结

本章学习了如何使用Item　Pipeline对爬取到的数据进行处理，先以一个简单的例子讲解了Item　Pipeline的应用场景以及具体使用，然后展示了Item Pipeline实际应用的两个例子。

第6章 使用LinkExtractor提取链接

在爬取一个网站时，想要爬取的数据通常分布在多个页面中，每个页面包含一部分数据以及到其他页面的链接，提取页面中数据的方法大家已经掌握，提取链接有使用Selector和使用LinkExtractor两种方法。

本章来学习如何提取页面中的链接。

1．使用Selector

因为链接也是页面中的数据，所以可以使用与提取数据相同的方法进行提取，在提取少量（几个）链接或提取规则比较简单时，使用Selector就足够了。

2．使用LinkExtractor

Scrapy提供了一个专门用于提取链接的类LinkExtractor，在提取大量链接或提取规则比较复杂时，使用LinkExtractor更加方便。

在第1章的example项目中使用了第一种方法提取下一页链接，回顾其中的代码片段：

```
class BooksSpider(scrapy.Spider):
    ...
def parse(self, response):
    ...
    # 提取链接
    # 下一页的url 在ul.pager > li.next > a 里面
    # 例如： <li class="next"><a href="catalogue/pag
    next_url = response.css('ul.pager li.next a::at
    if next_url:
        # 如果找到下一页的url，得到绝对路径，构造新的Reque
        next_url = response.urljoin(next_url)
        yield scrapy.Request(next_url, callback=sel
```

```
    ...
```

上述代码中，先使用CSS选择器选中包含下一页链接的a元素并获取其href属性，然后调用response.urljoin方法计算出绝对url地址，最后构造Request对象并提交。

第一种方法大家早已掌握，本章我们来学习如何使用LinkExtractor提取链接。

6.1 使用LinkExtractor

LinkExtractor的使用非常简单，通过一个例子进行讲解，使用LinkExtractor替代Selector完成BooksSpider提取链接的任务，代码如下：

```
from scrapy.linkextractors import LinkExtractor
class BooksSpider(scrapy.Spider):
    ...
    def parse(self, response):
        ...
        # 提取链接
        # 下一页的url 在ul.pager > li.next > a 里面
        # 例如: <li class="next"><a href="catalogue/
        le = LinkExtractor(restrict_css='ul.pager l
        links = le.extract_links(response)
        if links:
            next_url = links[0].url
            yield scrapy.Request(next_url, callback
```

对上述代码解释如下：

- 导入LinkExtractor，它位于scrapy.linkextractors模块。
- 创建一个LinkExtractor对象，使用一个或多个构造器参数描述提取规则，这里传递给restrict_css参数一个CSS选择器表达式。它描述出下一页链接所在的区域（在li.next下）。

- 调用LinkExtractor对象的extract_links方法传入一个Response对象，该方法依据创建对象时所描述的提取规则，在Response对象所包含的页面中提取链接，最终返回一个列表，其中的每一个元素都是一个Link对象，即提取到的一个链接。
- 由于页面中的下一页链接只有一个，因此用links[0]获取Link对象，Link对象的url属性便是链接页面的绝对url地址（无须再调用response.urljoin方法），用其构造Request对象并提交。

通过上面的例子，相信大家已经了解了使用LinkExtractor对象提取页面中链接的流程。

6.2　描述提取规则

接下来，我们来学习使用LinkExtractor的构造器参数描述提取规则。

为了在讲解过程中举例，首先制造一个实验环境，创建两个包含多个链接的HTML页面：

```
<!-- example1.html -->
<html>
   <body>
      <div id="top">
         <p>下面是一些站内链接</p>
         <a class="internal" href="/intro/install.
         <a class="internal" href="/intro/tutorial
         <a class="internal" href="../examples.htm
      </div>
      <div id="bottom">
         <p>下面是一些站外链接</p>
         <a href="http://stackoverflow.com/tags/sc
         <a href="https://github.com/scrapy/scrapy
      </div>
```

```
    </body>
</html>
<!-- example2.html -->
<html>
    <head>
        <script type='text/javascript' src='/js/app
        <script type='text/javascript' src='/js/app
    </head>
    <body>
        <a href="/home.html">主页</a>
        <a href="javascript:goToPage('/doc.html');
        <a href="javascript:goToPage('/example.html
    </body>
</html>
```

使用以上两个HTML文本构造两个Response对象：

```
>>> from scrapy.http import HtmlResponse
>>> html1 = open('exmaple1.html').read()
>>> html2 = open('exmaple2.html').read()
>>> response1 = HtmlResponse(url='http://example1.
>>> response2 = HtmlResponse(url='http://example2.
```

现在有了实验环境，先说明一种特例情况，LinkExtractor构造器的所有参数都有默认值，如果构造对象时不传递任何参数（使用默认值），就提取页面中所有链接。以下代码将提取页面example1.html中的所有链接：

```
>>> from scrapy.linkextractors import LinkExtracto
>>> le = LinkExtractor()
>>> links = le.extract_links(response1)
>>> [link.url for link in links]
['http://example1.com/intro/install.html',
'http://example1.com/intro/tutorial.html',
'http://example1.com/../examples.html',
'http://stackoverflow.com/tags/scrapy/info',
```

```
'https://github.com/scrapy/scrapy']
```

下面依次介绍LinkExtractor构造器的各个参数：

- allow

  接收一个正则表达式或一个正则表达式列表，提取绝对url与正则表达式匹配的链接，如果该参数为空（默认），就提取全部链接。

  示例　提取页面example1.html中路径以/intro开始的链接：

```
>>> from scrapy.linkextractors import LinkExtracto
>>> pattern = '/intro/.+\.html$'
>>> le = LinkExtractor(allow=pattern)
>>> links = le.extract_links(response1)
>>> [link.url for link in links]
['http://example1.com/intro/install.html',
'http://example1.com/intro/tutorial.html']
```

- deny

  接收一个正则表达式或一个正则表达式列表，与allow相反，排除绝对url与正则表达式匹配的链接。

  示例　提取页面example1.html中所有站外链接（即排除站内链接）：

```
>>> from scrapy.linkextractors import LinkExtracto
>>> from urllib.parse import urlparse
>>> pattern = patten = '^' + urlparse(response1.ur
>>> pattern
'^http://example1.com'
>>> le = LinkExtractor(deny=pattern)
>>> links = le.extract_links(response1)
>>> [link.url for link in links]
['http://stackoverflow.com/tags/scrapy/info',
```

```
 'https://github.com/scrapy/scrapy']
```

- allow_domains

  接收一个域名或一个域名列表，提取到指定域的链接。

  示例　提取页面example1.html中所有到github.com和stackoverflow.com这两个域的链接：

```
>>> from scrapy.linkextractors import LinkExtracto
>>> domains = ['github.com', 'stackoverflow.com']
>>> le = LinkExtractor(allow_domains=domains)
>>> links = le.extract_links(response1)
>>> [link.url for link in links]
['http://stackoverflow.com/tags/scrapy/info',
 'https://github.com/scrapy/scrapy']
```

- deny_domains

  接收一个域名或一个域名列表，与allow_domains相反，排除到指定域的链接。

  示例　提取页面example1.html中除了到github.com域以外的链接：

```
>>> from scrapy.linkextractors import LinkExtracto
>>> le = LinkExtractor(deny_domains='github.com')
>>> links = le.extract_links(response1)
>>> [link.url for link in links]
['http://example1.com/intro/install.html',
 'http://example1.com/intro/tutorial.html',
 'http://example1.com/../examples.html',
 'http://stackoverflow.com/tags/scrapy/info']
```

- restrict_xpaths

  接收一个XPath表达式或一个XPath表达式列表，提取XPath

表达式选中区域下的链接。

示例　提取页面example1.html中＜div id＝"top"＞元素下的链接：

```
>>> from scrapy.linkextractors import LinkExtracto
>>> le = LinkExtractor(restrict_xpaths='//div[@id=
>>> links = le.extract_links(response1)
>>> [link.url for link in links]
['http://example1.com/intro/install.html',
'http://example1.com/intro/tutorial.html',
'http://example1.com/../examples.html']
```

- restrict_css

  接收一个CSS选择器或一个CSS选择器列表，提取CSS选择器选中区域下的链接。

  示例　提取页面example1.html中＜div id＝"bottom"＞元素下的链接：

```
>>> from scrapy.linkextractors import LinkExtracto
>>> le = LinkExtractor(restrict_css='div#bottom')
>>> links = le.extract_links(response1)
>>> [link.url for link in links]
['http://stackoverflow.com/tags/scrapy/info',
 'https://github.com/scrapy/scrapy']
```

- tags

  接收一个标签（字符串）或一个标签列表，提取指定标签内的链接，默认为['a', 'area']。

- attrs

  接收一个属性（字符串）或一个属性列表，提取指定属性内的链接，默认为['href']。

示例　提取页面example2.html中引用JavaScript文件的链接：

```
>>> from scrapy.linkextractors import LinkExtracto
>>> le = LinkExtractor(tags='script', attrs='src')
>>> links = le.extract_links(response2)
>>> [link.url for link in links]
['http://example2.com/js/app1.js',
 'http://example2.com/js/app2.js']
```

- process_value

接收一个形如func(value)的回调函数。如果传递了该参数，LinkExtractor将调用该回调函数对提取的每一个链接（如a的href）进行处理，回调函数正常情况下应返回一个字符串（处理结果），想要抛弃所处理的链接时，返回None。

示例　在页面example2.html中，某些a的href属性是一段JavaScript代码，代码中包含了链接页面的实际url地址，此时应对链接进行处理，提取页面example2.html中所有实际链接：

```
>>> import re
>>> def process(value):
...        m = re.search("javascript:goToPage\('(.*
...        # 如果匹配，就提取其中url 并返回，不匹配则返回
...        if m:
...            value = m.group(1)
...        return value
…
>>> from scrapy.linkextractors import LinkExtracto
>>> le = LinkExtractor(process_value=process)
>>> links = le.extract_links(response2)
>>> [link.url for link in links]
['http://example2.com/home.html',
'http://example2.com/doc.html',
```

```
    'http://example2.com/example.html']
```

到此，我们介绍完了LinkExtractor构造器的各个参数，实际应用时可以同时使用一个或多个参数描述提取规则，这里不再举例。

6.3　本章小结

在Scrapy中，可以使用Selector或LinkExtractor提取页面中的链接，本章主要介绍后一种方法，先以一个案例展示了使用LinkExtractor提取链接的流程，然后详细讲解如何使用LinkExtractor的构造器参数描述提取规则。

第7章 使用Exporter导出数据

通过之前章节的学习，大家掌握了Scrapy中爬取数据、封装数据、处理数据的相关技术，本章我们来学习如何将爬取到的数据以某种数据格式保存到文件中，即导出数据。

在Scrapy中，负责导出数据的组件被称为Exporter（导出器），Scrapy内部实现了多个Exporter，每个Exporter实现一种数据格式的导出，支持的数据格式如下（括号中为相应的Exporter）：

（1）JSON (JsonItemExporter)

（2）JSON lines (JsonLinesItemExporter)

（3）CSV (CsvItemExporter)

（4）XML (XmlItemExporter)

（5）Pickle (PickleItemExporter)

（6）Marshal (MarshalItemExporter)

其中，前4种是极为常用的文本数据格式，而后两种是Python特有的。在大多数情况下，使用Scrapy内部提供的Exporter就足够了，需要以其他数据格式（上述6种以外）导出数据时，可以自行实现Exporter。

7.1　指定如何导出数据

在导出数据时，需向Scrapy爬虫提供以下信息：

- 导出文件路径。
- 导出数据格式（即选用哪个Exporter）。

可以通过以下两种方式指定爬虫如何导出数据：

（1）通过命令行参数指定。

（2）通过配置文件指定。

7.1.1　命令行参数

在运行scrapy crawl命令时，可以分别使用-o和-t参数指定导出文件路径以及导出数据格式。

在第1章example项目中，我们使用以下命令运行爬虫：

```
$ scrapy crawl books -o books.csv
...
$ head -10 books.csv # 查看文件开头的10 行
name,price
A Light in the Attic,£51.77
Tipping the Velvet,£53.74
Soumission,£50.10
Sharp Objects,£47.82
Sapiens: A Brief History of Humankind,£54.23
The Requiem Red,£22.65
The Dirty Little Secrets of Getting Your Dream Job
"The Coming Woman: A Novel Based on the Life of th
Woodhull",£17.93
The Boys in the Boat: Nine Americans and Their Epi
Olympics,£22.60
```

其中，-o　books.csv指定了导出文件的路径，在这里虽然没有使用-t参数指定导出数据格式，但Scrapy爬虫通过文件后缀名推断出我们想以csv作为导出数据格式。同样的道理，如果将参数改为-o books.json，Scrapy爬虫就会以json作为导出数据格式。

需要明确地指定导出数据格式时，使用-t参数，例如：

```
$ scrapy crawl books -t csv -o books1.data
...
```

```
$ scrapy crawl books -t json -o books2.data
...
$ scrapy crawl books -t xml -o books3.data
...
```

运行以上命令后，Scrapy爬虫会以-t参数中的数据格式字符串（如csv、json、xml）为键，在配置字典FEED_EXPORTERS中搜索Exporter，FEED_EXPORTERS的内容由以下两个字典的内容合并而成：

- 默认配置文件中的FEED_EXPORTERS_BASE。
- 用户配置文件中的FEED_EXPORTERS。

前者包含内部支持的导出数据格式，后者包含用户自定义的导出数据格式。以下是Scrapy源码中定义的FEED_EXPORTERS_BASE，它位于scrapy.settings.default_settings模块：

```
FEED_EXPORTERS_BASE = {
    'json': 'scrapy.exporters.JsonItemExporter',
    'jsonlines': 'scrapy.exporters.JsonLinesItemExp
    'jl': 'scrapy.exporters.JsonLinesItemExporter',
    'csv': 'scrapy.exporters.CsvItemExporter',
    'xml': 'scrapy.exporters.XmlItemExporter',
    'marshal': 'scrapy.exporters.MarshalItemExporte
    'pickle': 'scrapy.exporters.PickleItemExporter'
}
```

如果用户添加了新的导出数据格式（即实现了新的Exporter），可在配置文件settings.py中定义FEED_EXPORTERS，例如：

```
FEED_EXPORTERS = {'excel': 'my_project.my_exporter
```

另外，指定导出文件路径时，还可以使用%(name)s和%(time)s两个特殊变量：

- %(name)s:会被替换为Spider的名字。
- %(time)s:会被替换为文件创建时间。

请看一个例子，假设一个项目中有爬取书籍信息、游戏信息、新闻信息的3个Spider，分别名为'books'、'games'、'news'。对于任意Spider的任意一次爬取，都可以使用'export_data/%(name)s/%(time)s.csv'作为导出路径，Scrapy爬虫会依据Spider的名字和爬取的时间点创建导出文件：

```
$ scrapy crawl books -o 'export_data/%(name)s/%(time)s.csv'
...
$ scrapy crawl games -o 'export_data/%(name)s/%(time)s.csv'
...
$ scrapy crawl news -o 'export_data/%(name)s/%(time)s.csv'
...
$ scrapy crawl books -o 'export_data/%(name)s/%(time)s.csv'
...
$ tree export_data
export_data/
├── books
│   ├── 2017-03-06T02-31-57.csv
│   └── 2017-06-07T04-45-13.csv
├── games
│   └── 2017-04-05T01-43-01.csv
└── news
    └── 2017-05-06T09-44-06.csv
```

使用命令行参数指定如何导出数据很方便，但命令行参数只能指定导出文件路径以及导出数据格式，并且每次都在命令行里输入很长的参数让人很烦躁，使用配置文件可以弥补这些不足。

7.1.2 配置文件

接下来，我们学习在配置文件中指定如何导出数据。下面依次说明几个常用选项：

- FEED_URI

  导出文件路径。

```
FEED_URI = 'export_data/%(name)s.data'
```

- FEED_FORMAT

  导出数据格式。

```
FEED_FORMAT = 'csv'
```

- FEED_EXPORT_ENCODING

  导出文件编码（默认情况下json文件使用数字编码，其他使用utf-8编码）。

```
FEED_EXPORT_ENCODING = 'gbk'
```

- FEED_EXPORT_FIELDS

  导出数据包含的字段（默认情况下导出所有字段），并指定次序。

```
FEED_EXPORT_FIELDS = ['name', 'author', 'price']
```

- FEED_EXPORTERS

  用户自定义Exporter字典，添加新的导出数据格式时使用。

```
FEED_EXPORTERS = {'excel': 'my_project.my_exporter
```

7.2 添加导出数据格式

在某些需求下，我们想要添加新的导出数据格式，此时需要实现新的Exporter类。下面先参考Scrapy内部的Exporter类是如何实现的，然后自行实现一个Exporter。

7.2.1 源码参考

Scrapy内部的Exporter类在scrapy.exporters模块中实现，以下是其中的代码片段：

```
class BaseItemExporter(object):
   def __init__(self, **kwargs):
      self._configure(kwargs)
   def _configure(self, options, dont_fail=False):
      self.encoding = options.pop('encoding', None
      self.fields_to_export = options.pop('fields_
      self.export_empty_fields = options.pop('expo
      if not dont_fail and options:
         raise TypeError("Unexpected options: %s"
   def export_item(self, item):
      raise NotImplementedError
   def serialize_field(self, field, name, value):
      serializer = field.get('serializer', lambda
      return serializer(value)
   def start_exporting(self):
      pass
   def finish_exporting(self):
      pass
   def _get_serialized_fields(self, item, default_
       """Return the fields to export as an iterab
       (name, serialized_value)
       """
       if include_empty is None:
           include_empty = self.export_empty_field
       if self.fields_to_export is None:
           if include_empty and not isinstance(ite
              field_iter = six.iterkeys(item.field
           else:
              field_iter = six.iterkeys(item)
       else:
           if include_empty:
              field_iter = self.fields_to_export
           else:
```

```
                field_iter = (x for x in self.fields
        for field_name in field_iter:
            if field_name in item:
                field = {} if isinstance(item, dict)
                value = self.serialize_field(field,
            else:
                value = default_value
            yield field_name, value
# json
class JsonItemExporter(BaseItemExporter):
    def __init__(self, file, **kwargs):
        self._configure(kwargs, dont_fail=True)
        self.file = file
        kwargs.setdefault('ensure_ascii', not self.
        self.encoder = ScrapyJSONEncoder(**kwargs)
        self.first_item = True
    def start_exporting(self):
        self.file.write(b"[\n")
    def finish_exporting(self):
        self.file.write(b"\n]")
    def export_item(self, item):
        if self.first_item:
            self.first_item = False
        else:
            self.file.write(b',\n')
        itemdict = dict(self._get_serialized_fields
        data = self.encoder.encode(itemdict)
        self.file.write(to_bytes(data, self.encodin
# json lines
class JsonLinesItemExporter(BaseItemExporter):
    ...
# xml
class XmlItemExporter(BaseItemExporter):
    ...
# csv
class CsvItemExporter(BaseItemExporter):
    ...
...
```

其中的每一个Exporter都是BaseItemExporter的一个子类，BaseItemExporter定义了一些抽象接口待子类实现：

- export_item(self, item)

  负责导出爬取到的每一项数据，参数item为一项爬取到的数据，每个子类必须实现该方法。

- start_exporting(self)

  在导出开始时被调用，可在该方法中执行某些初始化工作。

- finish_exporting(self)

  在导出完成时被调用，可在该方法中执行某些清理工作。

以JsonItemExporter为例，其实现非常简单：

- 为了使最终导出结果是一个json中的列表，在start_exporting和finish_exporting方法中分别向文件写入b"[\n, b"\n]"。
- 在export_item方法中，调用self.encoder.encode方法将一项数据转换成json串（具体细节不再赘述），然后写入文件。

7.2.2 实现Exporter

接下来，我们参照JsonItemExporter的源码，在第1章example项目中实现一个能将数据以Excel格式导出的Exporter。

在项目中创建一个my_exporters.py（与settings.py同级目录），在其中实现ExcelItemExporter，代码如下：

```
from scrapy.exporters import BaseItemExporter
import xlwt
class ExcelItemExporter(BaseItemExporter):
   def __init__(self, file, **kwargs):
      self._configure(kwargs)
      self.file = file
      self.wbook = xlwt.Workbook()
      self.wsheet = self.wbook.add_sheet('scrapy')
```

```
        self.row = 0
    def finish_exporting(self):
        self.wbook.save(self.file)


    def export_item(self, item):
        fields = self._get_serialized_fields(item)
        for col, v in enumerate(x for _, x in fields
            self.wsheet.write(self.row, col, v)
        self.row += 1
```

解释上述代码如下：

- 这里使用第三方库xlwt将数据写入Excel文件。
- 在构造器方法中创建Workbook对象和Worksheet对象，并初始化用来记录写入行坐标的self.row。
- 在export_item方法中调用基类的_get_serialized_fields方法，获得item所有字段的迭代器，然后调用self.wsheet.write方法将各字段写入Excel表格。
- finish_exporting方法在所有数据都被写入Excel表格后被调用，在该方法中调用self.wbook.save方法将Excel表格写入Excel文件。

完成ExcelItemExporter后，在配置文件settings.py中添加如下代码：

```
FEED_EXPORTERS = {'excel': 'example.my_exporters.E
```

现在，可以使用ExcelItemExporter导出数据了，以-t excel为参数重新运行爬虫：

```
$ scrapy crawl books -t excel -o books.xls
```

图7-1所示为爬取完成后在Excel文件中观察到的结果。

	A	B
1	A Light in the Attic	£51.77
2	Tipping the Velvet	£53.74
3	Soumission	£50.10
4	Sharp Objects	£47.82
5	Sapiens: A Brief History of Humankind	£54.23
6	The Requiem Red	£22.65
7	The Dirty Little Secrets of Getting Your Dream Job	£33.34
8	The Coming Woman: A Novel Based on the Life of the Infamous Feminist, Victoria Woodhull	£17.93
9	The Boys in the Boat: Nine Americans and Their Epic Quest for Gold at the 1936 Berlin Olympics	£22.60
10	The Black Maria	£52.15
11	Starving Hearts (Triangular Trade Trilogy, #1)	£13.99
12	Shakespeare's Sonnets	£20.66
13	Set Me Free	£17.46
14	Scott Pilgrim's Precious Little Life (Scott Pilgrim #1)	£52.29
15	Rip it Up and Start Again	£35.02
16	Our Band Could Be Your Life: Scenes from the American Indie Underground, 1981-1991	£57.25
17	Olio	£23.88
18	Mesaerion: The Best Science Fiction Stories 1800-1849	£37.59
19	Libertarianism for Beginners	£51.33
20	It's Only the Himalayas	£45.17
21	In Her Wake	£12.84
22	How Music Works	£37.32
23	Foolproof Preserving: A Guide to Small Batch Jams, Jellies, Pickles, Condiments, and More: A Fo	£30.52
24	Chase Me (Paris Nights #2)	£25.27
25	Black Dust	£34.53
26	Birdsong: A Story in Pictures	£54.64
27	America's Cradle of Quarterbacks: Western Pennsylvania's Football Factory from Johnny Unitas t	£22.50
28	Aladdin and His Wonderful Lamp	£53.13
29	Worlds Elsewhere: Journeys Around Shakespeare's Globe	£40.30
30	Wall and Piece	£44.18
31	The Four Agreements: A Practical Guide to Personal Freedom	£17.66
32	The Five Love Languages: How to Express Heartfelt Commitment to Your Mate	£31.05
33	The Elephant Tree	£23.82
34	The Bear and the Piano	£36.89

图7-1

如上所示，我们成功地使用ExcelItemExporter将爬取到的数据存入了Excel文件中。

7.3 本章小结

本章学习了在Scrapy中如何使用Exporter将爬取到的数据导出到文件，首先介绍使用命令行参数以及配置文件指定如何导出数据的方法，然后参考Scrapy内部Exporter的源码实现了一个能将数据导出到Excel文件的Exporter。

第8章
项目练习

通过之前章节的学习，大家掌握了编写Scrapy爬虫的基础知识，这一章我们运用之前所学进行实战项目练习。

在第1章的example项目中，我们爬取了http://books.toscrape.com网站中的书籍信息，但仅从每一个书籍列表页面爬取了书的名字和价格信息，如图8-1所示。

图8-1

通常，实际应用需求并不会这么简单，可能需要获取每本书的更多信息，在具体一本书的页面中可以找到更多的信息，点击第一本书的链接，将看到如图8-2所示的页面。

图8-2

如上所示，在一本书的页面中可以获取以下信息：

- 书名√
- 价格√
- 评价等级√
- 书籍简介
- 产品编码√
- 产品类型
- 税价
- 库存量√
- 评价数量√

下面我们新建一个Scrapy项目，爬取每一本书更多的信息（只爬取其中打对号的信息）。

8.1　项目需求

下面爬取http://books.toscrape.com网站中的书籍信息。

（1）其中每一本书的信息包括：

- 书名
- 价格
- 评价等级
- 产品编码
- 库存量
- 评价数量

（2）将爬取的结果保存到csv文件中。

8.2 页面分析

首先，我们对一本书的页面进行分析。在进行页面分析时，除了之前使用过的Chrome开发者工具外，另一个常用的工具是scrapy shell ＜URL＞命令，它使用户可以在交互式命令行下操作一个Scrapy爬虫，通常我们利用该工具进行前期爬取实验，从而提高开发效率。

接下来分析第一本书的页面，以页面的url地址为参数运行scrapy shell命令：

```
    $ scrapy shell http://books.toscrape.com/catal
    scrapy shell http://books.toscrape.com/catalog
    2017-03-03 09:17:01 [scrapy] INFO: Scrapy 1.3.
    2017-03-03 09:17:01 [scrapy] INFO: Overridden
'DUPEFILTER_CLASS': 'scrapy.dupefilters.BaseDupeFi
    2017-03-03 09:17:01 [scrapy] INFO: Enabled ext
    ['scrapy.extensions.corestats.CoreStats',
     'scrapy.extensions.telnet.TelnetConsole']
    2017-03-03 09:17:01 [scrapy] INFO: Enabled dow
['scrapy.downloadermiddlewares.httpauth.HttpAuthMi
 'scrapy.downloadermiddlewares.downloadtimeout.Dow
 'scrapy.downloadermiddlewares.defaultheaders.Defa
 'scrapy.downloadermiddlewares.useragent.UserAgent
 'scrapy.downloadermiddlewares.retry.RetryMiddlewa
 'scrapy.downloadermiddlewares.redirect.MetaRefres
 'scrapy.downloadermiddlewares.httpcompression.Htt
 'scrapy.downloadermiddlewares.redirect.RedirectMi
 'scrapy.downloadermiddlewares.cookies.CookiesMidd
 'scrapy.downloadermiddlewares.chunked.ChunkedTran
 'scrapy.downloadermiddlewares.stats.DownloaderSta
2017-03-03 09:17:01 [scrapy] INFO: Enabled spider
['scrapy.spidermiddlewares.httperror.HttpErrorMidd
 'scrapy.spidermiddlewares.offsite.OffsiteMiddlewa
 'scrapy.spidermiddlewares.referer.RefererMiddlewa
 'scrapy.spidermiddlewares.urllength.UrlLengthMidd
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2017-03-03 09:17:01 [scrapy] INFO: Enabled item pi
```

```
[]
2017-03-03 09:17:01 [scrapy] DEBUG: Telnet console
2017-03-03 09:17:01 [scrapy] INFO: Spider opened
2017-03-03 09:17:01 [scrapy] DEBUG: Crawled (200)
2017-03-03 09:17:02 [traitlets] DEBUG: Using defau
2017-03-03 09:17:02 [traitlets] DEBUG: Using defau
[s] Available Scrapy objects:
[s]   scrapy      scrapy module (contains scrapy.Re
[s]   crawler
[s]   item        {}
[s]   request
[s]   response    <200 http://books.toscrape.com/ca
[s]   settings
[s]   spider
[s] Useful shortcuts:
[s]   shelp()            Shell help (print this hel
[s]   fetch(req_or_url) Fetch request (or URL) and
[s]   view(response)     View response in a browser
>>>
```

运行这条命令后，scrapy shell会使用url参数构造一个Request对象，并提交给Scrapy引擎，页面下载完成后，程序进入一个python shell当中，在此环境中已经创建好了一些变量（对象和函数），以下几个最为常用：

- request

  最近一次下载对应的Request对象。

- response

  最近一次下载对应的Response对象。

- fetch(req_or_url)

  该函数用于下载页面，可传入一个Request对象或url字符串，调用后会更新变量request和response。

- view(response)

  该函数用于在浏览器中显示response中的页面。

接下来，在scrapy shell中调用view函数，在浏览器中显示response所包含的页面：

```
>>> view(response)
```

可能在很多时候，使用view函数打开的页面和在浏览器直接输入url打开的页面看起来是一样的，但需要知道的是，前者是由Scrapy爬虫下载的页面，而后者是由浏览器下载的页面，有时它们是不同的。在进行页面分析时，使用view函数更加可靠。下面使用Chrome审查元素工具分析页面，如图8-3所示。

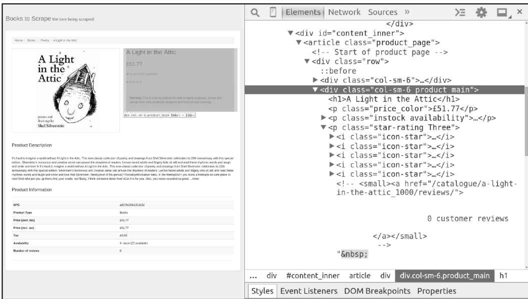


图8-3

从图8-3中看出，我们可在<div class="col-sm-6 product_main">中提取书名、价格、评价等级，在scrapy shell中尝试提取这些信息，如图8-4所示。

```
>>> sel = response.css('div.product_main')
>>> sel.xpath('./h1/text()').extract_first()
'A Light in the Attic'
>>> sel.css('p.price_color::text').extract_first()
'£51.77'
>>> sel.css('p.star-rating::attr(class)').re_first
'Three'
```



图8-4

另外，可在页面下端位置的<table class="table table-striped">中提取产品编码、库存量、评价数量，在scrapy shell中尝试提取这些信息：

```
>>> sel = response.css('table.table.table-striped'
>>> sel.xpath('(.//tr)[1]/td/text()').extract_firs
'a897fe39b1053632'
>>> sel.xpath('(.//tr)[last()-1]/td/text()').re_fi
'22'
>>> sel.xpath('(.//tr)[last()]/td/text()').extract
'0'
```

分析完书籍页面后，接着分析如何在书籍列表页面中提取每一个书籍页面的链接。在scrapy shell中，先调用fetch函数下载第一个书籍列表页面（http://books.toscrape.com/），下载完成后再调用view函数在浏览器中查看页面，如图8-5所示。

```
>>> fetch('http://books.toscrape.com/')
[scrapy] DEBUG: Crawled (200) <GET http://books.tc
>>> view(response)
```

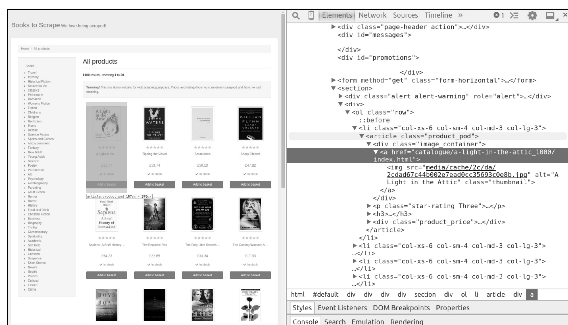


图8-5

每个书籍页面的链接可以在每个<article class="product_pod">中找到，在scrapy shell中使用LinkExtractor提取这些链接：

```
    >>> from scrapy.linkextractors import LinkExtr
    >>> le = LinkExtractor(restrict_css='article.p
    >>> le.extract_links(response)
    [Link(url='http://books.toscrape.com/catalogue
fragment='', nofollow=False),
    Link(url='http://books.toscrape.com/catalogue/
nofollow=False),
    Link(url='http://books.toscrape.com/catalogue/
nofollow=False),
     Link(url='http://books.toscrape.com/catalogue
nofollow=False),
     Link(url='http://books.toscrape.com/catalogue
text='', fragment='', nofollow=False),
     Link(url='http://books.toscrape.com/catalogue
nofollow=False),
     Link(url='http://books.toscrape.com/catalogue
ndex.html', text='', fragment='', nofollow=False),
     Link(url='http://books.toscrape.com/catalogue
famous-feminist-victoria-woodhull_993/index.html',
     Link(url='http://books.toscrape.com/catalogue
st-for-gold-at-the-1936-berlin-olympics_992/index.
     Link(url='http://books.toscrape.com/catalogue
nofollow=False),
```

```
        Link(url='http://books.toscrape.com/catalogue
ml', text='', fragment='', nofollow=False),
        Link(url='http://books.toscrape.com/catalogue
fragment='', nofollow=False),
        Link(url='http://books.toscrape.com/catalogue
nofollow=False),
        Link(url='http://books.toscrape.com/catalogue
ndex.html', text='', fragment='', nofollow=False),
        Link(url='http://books.toscrape.com/catalogue
fragment='', nofollow=False),
        Link(url='http://books.toscrape.com/catalogue
-indie-underground-1981-1991_985/index.html', text
        Link(url='http://books.toscrape.com/catalogue
nofollow=False),
        Link(url='http://books.toscrape.com/catalogue
983/index.html', text='', fragment='', nofollow=Fa
        Link(url='http://books.toscrape.com/catalogue
fragment='', nofollow=False),
        Link(url='http://books.toscrape.com/catalogue
fragment='', nofollow=False)]
```

到此，页面分析的工作已经完成了。

8.3 编码实现

首先创建一个Scrapy项目，取名为toscrape_book。

```
$ scrapy startproject toscrape_book
```

通常，我们不需要手工创建Spider文件以及Spider类，可以使用scrapy genspider＜SPIDER_NAME＞ ＜DOMAIN＞命令生成（根据模板）它们，该命令的两个参数分别是Spider的名字和所要爬取的域（网站）：

```
$ cd toscrape_book
```

```
$ scrapy genspider books books.toscrape.com
```

运行后，scrapy genspider命令创建了文件toscrape_book/spiders/books.py，并在其中创建了一个BooksSpider类，代码如下：

```
# -*- coding: utf-8 -*-
import scrapy


class BooksSpider(scrapy.Spider):
    name = "books"
    allowed_domains = ["books.toscrape.com"]
    start_urls = ['http://books.toscrape.com/']


    def parse(self, response):
        pass
```

实现Spider之前，先定义封装书籍信息的Item类，在toscrape_book/items.py中添加如下代码：

```
class BookItem(scrapy.Item):
    name = scrapy.Field()                # 书名
    price = scrapy.Field()               # 价格
    review_rating = scrapy.Field()       # 评价等级，1～5
    review_num = scrapy.Field()          # 评价数量
    upc = scrapy.Field()                 # 产品编码
    stock = scrapy.Field()               # 库存量
```

接下来，按以下5步完成BooksSpider。

步骤 **01** 继承Spider创建BooksSpider类（已完成）。

步骤 **02** 为Spider取名（已完成）。

步骤 **03** 指定起始爬取点（已完成）。

步骤　**04**　实现书籍列表页面的解析函数。

步骤　**05**　实现书籍页面的解析函数。

其中前3步已经由scrapy　genspider命令帮我们完成，不需做任何修改。

第4步和第5步的工作是实现两个页面解析函数，因为起始爬取点是一个书籍列表页面，我们就将parse方法作为书籍列表页面的解析函数，另外，还需要添加一个parse_book方法作为书籍页面的解析函数，代码如下：

```
class BooksSpider(scrapy.Spider):
    name = "books"
    allowed_domains = ["books.toscrape.com"]
    start_urls = ['http://books.toscrape.com/']

    # 书籍列表页面的解析函数
    def parse(self, response):
        pass

    # 书籍页面的解析函数
    def parse_book(self, reponse):
        pass
```

先来完成第4步，实现书籍列表页面的解析函数（parse方法），需要完成以下两个任务：

（1）提取页面中每一个书籍页面的链接，用它们构造Request对象并提交。

（2）提取页面中下一个书籍列表页面的链接，用其构造Request对象并提交。

提取链接的具体细节在页面分析时已经讨论过，实现代码如下：

```
class BooksSpider(scrapy.Spider):
   name = "books"
   allowed_domains = ["books.toscrape.com"]
   start_urls = ['http://books.toscrape.com/']

   # 书籍列表页面的解析函数
   def parse(self, response):
   # 提取书籍列表页面中每本书的链接
   le = LinkExtractor(restrict_css='article.produc
   for link in le.extract_links(response):
       yield scrapy.Request(link.url, callback=sel

   # 提取"下一页"的链接
   le = LinkExtractor(restrict_css='ul.pager li.ne
   links = le.extract_links(response)
   if links:
       next_url = links[0].url
       yield scrapy.Request(next_url, callback=sel

# 书籍页面的解析函数
def parse_book(self, response):
   pass
```

最后完成第5步，实现书籍页面的解析函数（parse_book方法），只需提取书籍信息存入BookItem对象即可。同样，提取书籍信息的细节也在页面分析时讨论过，最终完成代码如下：

```
import scrapy
from scrapy.linkextractors import LinkExtractor
from ..items import BookItem

class BooksSpider(scrapy.Spider):
   name = "books"
```

```
    allowed_domains = ["books.toscrape.com"]
    start_urls = ['http://books.toscrape.com/']


    def parse(self, response):
        le = LinkExtractor(restrict_css='article.pr
        for link in le.extract_links(response):
            yield scrapy.Request(link.url, callback


        le = LinkExtractor(restrict_css='ul.pager l
        links = le.extract_links(response)
        if links:
            next_url = links[0].url
            yield scrapy.Request(next_url, callback
def parse_book(self, response):
    book = BookItem()
    sel = response.css('div.product_main')
    book['name'] = sel.xpath('./h1/text()').extract
    book['price'] = sel.css('p.price_color::text').
    book['review_rating'] = sel.css('p.star-rating:
                            .re_first('star-rating


    sel = response.css('table.table.table-striped')
    book['upc'] = sel.xpath('(.//tr)[1]/td/text()')
    book['stock'] = sel.xpath('(.//tr)[last()-1]/td
                    .re_first('\((\d+) available\
    book['review_num'] = sel.xpath('(.//tr)[last()]


    yield book
```

完成代码后，运行爬虫并观察结果：

```
$ scrapy crawl books -o books.csv --nolog
$ cat -n books.csv
    1   name,stock,price,review_num,review_rating,
```

```
   2   Scott Pilgrim's Precious Little Life,19,£
   3   It's Only the Himalayas,19,£45.17,0,Two,a
   4   Olio,19,£23.88,0,One,feb7cc7701ecf901
   5   Rip it Up and Start Again,19,£35.02,0,Fiv

   ... 省略中间输出 ...

 999   Bright Lines,1,£39.07,0,Five,230ac636ea0e
1000  Jurassic Park (Jurassic Park #1),3,£44.97,
1001  Into the Wild,3,£56.70,0,Five,a7c3f1010d64
```

从以上结果中看出，我们成功地爬取了网站中1000本书的详细信息，但也有让人不满意的地方，比如csv文件中各列的次序是随机的，看起来比较混乱，可在配置文件settings.py中使用FEED_EXPORT_FIELDS指定各列的次序：

```
FEED_EXPORT_FIELDS = ['upc', 'name', 'price', 'stc
```

另外，结果中评价等级字段的值是One、Two、Three……这样的单词，而不是阿拉伯数字，阅读起来不是很直观。下面实现一个Item Pipeline，将评价等级字段由单词映射到数字（或许这样简单的需求使用Item Pipeline有点大材小用，主要目的是带领大家复习之前所学的知识）。在pipelines.py中实现BookPipeline，代码如下：

```
class BookPipeline(object):
   review_rating_map = {
      'One':     1,
      'Two':     2,
      'Three':  3,
      'Four':   4,
      'Five':   5,
   }
```

```
    def process_item(self, item, spider):
        rating = item.get('review_rating')
        if rating:
            item['review_rating'] = self.review_ratin

        return item
```

在配置文件settings.py中启用BookPipeline：

```
ITEM_PIPELINES = {
  'toscrape_book.pipelines.BookPipeline': 300,
}
```

重新运行爬虫，并观察结果：

```
$ scrapy crawl books -o books.csv
...
$ cat -n books.csv
    1   upc,name,price,stock,review_rating,review_
    2   a897fe39b1053632,A Light in the Attic,£51
    3   3b1c02bac2a429e6,Scott Pilgrim's Precious
    4   a22124811bfa8350,It's Only the Himalayas,£
    5   feb7cc7701ecf901,Olio,£23.88,19,1,0

    ... 省略中间输出 ...

  999   91eb9605998a7c03,"The Sandman, Vol. 3: Dre
 1000   f06039c29b5891fa,The Silkworm (Cormoran St
 1001   476c7972e9b41891,The Last Painting of Sara
```

此时，各字段已按指定次序排列，并且评价等级字段的值是我们所期望的阿拉伯数字。

到此为止，整个项目完成了。

8.4　本章小结

本章是基础篇的最后一章，通过一个Scrapy爬虫项目复习了之前章节所学的知识，现在，大家已经能够编写一个一般任务的Scrapy爬虫了，可以通过更多的实战项目进行练习，在后面的章节中，我们将会学习一些高级话题。

第9章 下载文件和图片

在之前的章节中，我们学习了从网页中爬取信息的方法，这只是爬虫最典型的一种应用，除此之外，下载文件也是实际应用中很常见的一种需求，例如使用爬虫爬取网站中的图片、视频、WORD文档、PDF文件、压缩包等。本章来学习在Scrapy中如何下载文件和图片。

9.1 FilesPipeline和ImagesPipeline

Scrapy框架内部提供了两个Item Pipeline，专门用于下载文件和图片：

- FilesPipeline
- ImagesPipeline

我们可以将这两个Item Pipeline看作特殊的下载器，用户使用时只需要通过item的一个特殊字段将要下载文件或图片的url传递给它们，它们会自动将文件或图片下载到本地，并将下载结果信息存入item的另一个特殊字段，以便用户在导出文件中查阅。下面详细介绍如何使用它们。

9.1.1 FilesPipeline使用说明

通过一个简单的例子讲解FilesPipeline的使用，在如下页面中可以下载多本PDF格式的小说：

```
<html>
   <body>
      ...
      <a href='/book/sg.pdf'>下载《三国演义》</a>
      <a href='/book/shz.pdf'>下载《水浒传》</a>
```

```
        <a href='/book/hlm.pdf'>下载《红楼梦》</a>
        <a href='/book/xyj.pdf'>下载《西游记》</a>
        ...
    </body>
</html>
```

使用FilesPipeline下载页面中所有PDF文件，可按以下步骤进行：

步骤 **01** 在配置文件settings.py中启用FilesPipeline，通常将其置于其他Item Pipeline之前：

```
ITEM_PIPELINES = {'scrapy.pipelines.files.FilesPip
```

步骤 **02** 在配置文件settings.py中，使用FILES_STORE指定文件下载目录，如：

```
FILES_STORE = '/home/liushuo/Download/scrapy'
```

步骤 **03** 在Spider解析一个包含文件下载链接的页面时，将所有需要下载文件的url地址收集到一个列表，赋给item的file_urls字段（item['file_urls']）。FilesPipeline在处理每一项item时，会读取item['file_urls']，对其中每一个url进行下载，Spider示例代码如下：

```
class DownloadBookSpider(scrapy.Spider):
    ...

    def parse(response):
        item = {}
        # 下载列表
        item['file_urls'] = []
```

```
    for url in response.xpath('//a/@href').extract():
        download_url = response.urljoin(url)
        # 将url 填入下载列表
        item['file_urls'].append(download_url)


    yield item
```

当FilesPipeline下载完item['file_urls']中的所有文件后，会将各文件的下载结果信息收集到另一个列表，赋给item的files字段（item['files']）。下载结果信息包括以下内容：

- Path文件下载到本地的路径（相对于FILES_STORE的相对路径）。
- Checksum文件的校验和。
- url文件的url地址。

9.1.2 ImagesPipeline使用说明

图片也是文件，所以下载图片本质上也是下载文件，ImagesPipeline是FilesPipeline的子类，使用上和FilesPipeline大同小异，只是在所使用的item字段和配置选项上略有差别，如表9-1所示。

表9-1 ImagesPipeline和FilesPipeline

	FilesPipeline	ImagesPipeline
导入路径	scrapy.pipelines.files.FilesPipeline	scrapy.pipelines.images.ImagesPipeline
Item 字段	file_urls, files	image_urls, images
下载目录	FILES_STORE	IMAGES_STORE

ImagesPipeline在FilesPiplelline的基础上针对图片增加了一些特有的功能：

- 为图片生成缩略图

  开启该功能，只需在配置文件settings.py中设置IMAGES_THUMBS，它是一个字典，每一项的值是缩略图的

尺寸，代码如下：

```
IMAGES_THUMBS = {
   'small': (50, 50),
   'big': (270, 270),
}
```

开启该功能后，下载一张图片时，本地会出现3张图片（1张原图片，2张缩略图），路径如下：

```
[IMAGES_STORE]/full/63bbfea82b8880ed33cdb762aa11fa
[IMAGES_STORE]/thumbs/small/63bbfea82b8880ed33cdb7
[IMAGES_STORE]/thumbs/big/63bbfea82b8880ed33cdb762
```

- 过滤掉尺寸过小的图片

  开启该功能，需在配置文件settings.py中设置IMAGES_MIN_WIDTH和IMAGES_MIN_HEIGHT，它们分别指定图片最小的宽和高，代码如下：

```
IMAGES_MIN_WIDTH = 110
IMAGES_MIN_HEIGHT = 110
```

开启该功能后，如果下载了一张105×200的图片，该图片就会被抛弃掉，因为它的宽度不符合标准。

9.2　项目实战：爬取matplotlib例子源码文件

下面我们来完成一个使用FilesPipeline下载文件的实战项目。matplotlib是一个非常著名的Python绘图库，广泛应用于科学计算和数据分析等领域。在matplotlib网站上提供了许多应用例子代码，在浏览器中访问http://matplotlib.org/examples/index.html，可看到图

9-1所示的例子列表页面。

其中有几百个例子，被分成多个类别，单击第一个例子，进入其页面，如图9-2所示。

用户可以在每个例子页面中阅读源码，也可以点击页面中的source code按钮下载源码文件。如果我们想把所有例子的源码文件都下载到本地，可以编写一个爬虫程序完成这个任务。

9.2.1 项目需求

下载http://matplotlib.org网站中所有例子的源码文件到本地。

图9-1

图9-2

9.2.2　页面分析

先来看如何在例子列表页面http://matplotlib.org/examples/index.html中获取所有例子页面的链接。使用scrapy shell命令下载页面，然后调用view函数在浏览器中查看页面，如图9-3所示。

```
$ scrapy shell http://matplotlib.org/examples/inde
...
>>> view(response)
```

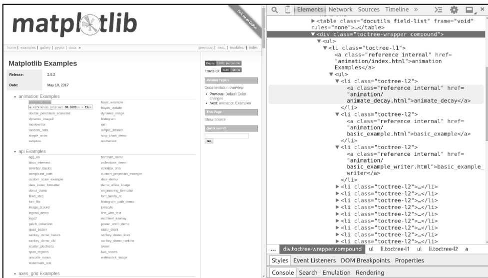


图9-3

观察发现，所有例子页面的链接都在<div class="toctree-wrapper compound">下的每一个<li class="toctree-l2">中，例如：

```
<a class="reference internal" href="animation/anim
```

使用LinkExtractor提取所有例子页面的链接，代码如下：

```
>>> from scrapy.linkextractors import LinkExtracto
>>> le = LinkExtractor(restrict_css='div.toctree-w
>>> links = le.extract_links(response)
>>> [link.url for link in links]
['http://matplotlib.org/examples/animation/animate
```

```
'http://matplotlib.org/examples/animation/basic_ex
'http://matplotlib.org/examples/animation/basic_ex
 'http://matplotlib.org/examples/animation/bayes_u
 'http://matplotlib.org/examples/animation/double_
 'http://matplotlib.org/examples/animation/dynamic
 'http://matplotlib.org/examples/animation/dynamic
 'http://matplotlib.org/examples/animation/histogr
 'http://matplotlib.org/examples/animation/moviewr
 'http://matplotlib.org/examples/animation/rain.ht
 'http://matplotlib.org/examples/animation/random_
 'http://matplotlib.org/examples/animation/simple_
 'http://matplotlib.org/examples/animation/simple_
 'http://matplotlib.org/examples/animation/strip_c
 'http://matplotlib.org/examples/animation/subplot
 'http://matplotlib.org/examples/animation/unchain
 'http://matplotlib.org/examples/api/agg_oo.html',
 'http://matplotlib.org/examples/api/barchart_demo
 'http://matplotlib.org/examples/api/bbox_intersec
 ...
 'http://matplotlib.org/examples/user_interfaces/s
 'http://matplotlib.org/examples/user_interfaces/t
 'http://matplotlib.org/examples/user_interfaces/w
 'http://matplotlib.org/examples/widgets/buttons.h
 'http://matplotlib.org/examples/widgets/check_but
 'http://matplotlib.org/examples/widgets/cursor.ht
 'http://matplotlib.org/examples/widgets/lasso_sel
 'http://matplotlib.org/examples/widgets/menu.html
 'http://matplotlib.org/examples/widgets/multicurs
 'http://matplotlib.org/examples/widgets/radio_but
 'http://matplotlib.org/examples/widgets/rectangle
 'http://matplotlib.org/examples/widgets/slider_de
 'http://matplotlib.org/examples/widgets/span_sele
>>> len(links)
507
```

例子列表页面分析完毕，总共找到了507个例子。

接下来分析例子页面。调用fetch函数下载第一个例子页面，并调用view函数在浏览器中查看页面，如图9-4所示。

```
>>> fetch('http://matplotlib.org/examples/animatic
...
>>> view(response)
```



图9-4

在一个例子页面中，例子源码文件的下载地址可在＜a class＝"reference external"＞中找到：

```
>>> href = response.css('a.reference.external::att
>>> href
'animate_decay.py'
>>> response.urljoin(href)
'http://matplotlib.org/examples/animation/animate_
```

到此，页面分析的工作完成了。

9.2.3　编码实现

接下来，我们按以下4步完成该项目：

（1）创建Scrapy项目，并使用scrapy　genspider命令创建Spider。

（2）在配置文件中启用FilesPipeline，并指定文件下载目录。

（3）实现ExampleItem（可选）。

（4）实现ExamplesSpider。

步骤　**01**　首先创建Scrapy项目，取名为matplotlib_examples，再使用scrapy genspider命令创建Spider：

```
$ scrapy startproject matplotlib_examples
$ cd matplotlib_examples
$ scrapy genspider examples matplotlib.org
```

步骤　**02**　在配置文件settings.py中启用FilesPipeline，并指定文件下载目录，代码如下：

```
ITEM_PIPELINES = {
  'scrapy.pipelines.files.FilesPipeline': 1,
}
FILES_STORE = 'examples_src'
```

步骤　**03**　实现ExampleItem，需定义file_urls和files两个字段，在items.py中完成如下代码：

```
class ExampleItem(scrapy.Item):
   file_urls = scrapy.Field()
   files = scrapy.Field()
```

步骤　**04**　实现ExamplesSpider。首先设置起始爬取点：

```
import scrapy


class ExamplesSpider(scrapy.Spider):
   name = "examples"
   allowed_domains = ["matplotlib.org"]
   start_urls = ['http://matplotlib.org/examples/i
```

```
    def parse(self, response):
        pass
```

parse方法是例子列表页面的解析函数，在该方法中提取每个例子页面的链接，用其构造Request对象并提交，提取链接的细节已在页面分析时讨论过，实现parse方法的代码如下：

```
import scrapy
from scrapy.linkextractors import LinkExtractor


class ExamplesSpider(scrapy.Spider):
    name = "examples"
    allowed_domains = ["matplotlib.org"]
    start_urls = ['http://matplotlib.org/examples/i
def parse(self, response):


    le = LinkExtractor(restrict_css='div.toctree-wr
                                     deny='/index.html$')
    print(len(le.extract_links(response)))
    for link in le.extract_links(response):
        yield scrapy.Request(link.url, callback=sel


def parse_example(self, response):
    pass
```

上面代码中，我们将例子页面的解析函数设置为parse_example方法，下面来实现这个方法。例子页面中包含了例子源码文件的下载链接，在parse_example方法中获取源码文件的url，将其放入一个列表，赋给ExampleItem的file_urls字段。实现parse_example方法的代码如下：

```
import scrapy
from scrapy.linkextractors import LinkExtractor
from ..items import ExampleItem
class ExamplesSpider(scrapy.Spider):
   name = "examples"
   allowed_domains = ["matplotlib.org"]
   start_urls = ['http://matplotlib.org/examples/i
   def parse(self, response):
      le = LinkExtractor(restrict_css='div.toctree
                             deny='/index.html$')
      print(len(le.extract_links(response)))
      for link in le.extract_links(response):
         yield scrapy.Request(link.url, callback=s
   def parse_example(self, response):
      href = response.css('a.reference.external::a
      url = response.urljoin(href)
      example = ExampleItem()
      example['file_urls'] = [url]
      return example
```

编码完成后，运行爬虫，并观察结果：

```
$ scrapy crawl examples -o examples.json
...
$ ls
examples.json  examples_src  matplotlib_examples
```

运行结束后，在文件examples.json中可以查看到文件下载结果信息：

```
     $ cat examples.json
     [
     {"file_urls": ["http://matplotlib.org/mpl_exa
"http://matplotlib.org/mpl_examples/axes_grid/demo
"502d1cd62086fb1d4de033cef2e495c0", "path":
```

```
"full/d9b551310a6668ccf43871e896f2fe6e0228567d.py"
     {"file_urls": ["http://matplotlib.org/mpl_exa
[{"url": "http://matplotlib.org/mpl_examples/axes_
"5cb91103f11079b40400afc0c1f4a508", "path":
"full/366386c23c5b715c49801efc7f8d55d2c74252e2.py"
     {"file_urls":
["http://matplotlib.org/mpl_examples/axes_grid/mak
[{"url": "http://matplotlib.org/mpl_examples/axes_
"checksum": "dcf561f97ab0905521c1957cacd2da00", "p
"full/919cbbe6d725237e3b6051f544f6109e7189b4fe.py"
     ...省略部分内容...
     {"file_urls": ["http://matplotlib.org/mpl_exa
[{"url": "http://matplotlib.org/mpl_examples/api/c
"bde485f9d5ceb4b4cc969ef692df5eee", "path":
"full/d56af342d7130ddd9dbf55c00664eae9a432bf70.py"
     {"file_urls": ["http://matplotlib.org/example
"http://matplotlib.org/examples/animation/dynamic_
"98b6a6021ba841ef4a2cd36c243c516d", "path":
"full/fe635002562e8685583c1b35a8e11e8cde0a6321.py"
     {"file_urls": ["http://matplotlib.org/example
"http://matplotlib.org/examples/animation/basic_ex
"1d4afc0910f6abc519e6ecd32c66896a", "path":
"full/083c113c1dac96bbc74adfc5b08cad68ec9c16db.py"
```

再来查看文件下载目录exmaples_src：

```
$ tree examples_src
examples_src
└─── full
    ├─── 006947659b6bd35d3f5a27df2bc36f81f1170e2e.py
    ├─── 00f4d142b951f0727f1d549b1d17574b5f841776.py
    ├─── 0100763ca91d69e64085cf770bf2a6f9d0c0136d.py
    ├─── 019bbcfcf06fb55ab46a1e52c71f56804bd06047.py
    ├─── 020bc2c11d2b0768a6df86a3b9dacf7adecf442d.py
    ├─── 028c0be66575ba9ab770330f7752e8e670308813.py

   ... 省略部分输出 ...

    ├─── fe7e5882c83d7c7c020bf9f3ae5570172a5fd3a6.py
    ├─── ffcb2f1e36fed60034a5d961674a0218857057b6.py
    ├─── ffdad211ab38c4f0ecba1857d11c7507b8e43ca8.py
    └─── fff6a4cf0d0261d63c79ba3606b195c8764ffc11.py

1 directory, 507 files
```

如上所示，507个源码文件被下载到了examples_src/full目录下，并且每个文件的名字都是一串长度相等的奇怪数字，这些数字是下载

文件url的sha1散列值。例如，某文件url为：

```
http://matplotlib.org/mpl_examples/axes_grid/demo_
```

该url的sha1散列值为：

```
d9b551310a6668ccf43871e896f2fe6e0228567d
```

那么该文件的存储路径为：

```
# [FILES_STORE]/full/[SHA1_HASH_VALUE].py
examples_src/full/d9b551310a6668ccf43871e896f2fe6e
```

这种命名方式可以防止重名的文件相互覆盖，但这样的文件名太不直观了，无法从文件名了解文件内容，我们期望把这些例子文件按照类别下载到不同目录下，为完成这个任务，可以写一个单独的脚本，依据examples.json文件中的信息将文件重命名，也可以修改FilesPipeline为文件命名的规则，这里采用后一种方式。

阅读FilesPipeline的源码发现，原来是其中的file_path方法决定了文件的命名，相关代码如下：

```
class FilesPipeline(MediaPipeline):
   ...
   def file_path(self, request, response=None, inf
   ...
   # check if called from file_key with url as fir
   if not isinstance(request, Request):
       _warn()
       url = request
   else:
       url = request.url
```

```
        # detect if file_key() method has been overridd
        if not hasattr(self.file_key, '_base'):
            _warn()
            return self.file_key(url)
        ## end of deprecation warning block
        media_guid = hashlib.sha1(to_bytes(url)).hexdig
        media_ext = os.path.splitext(url)[1]
        return 'full/%s%s' % (media_guid, media_ext)
    ...
```

现在，我们实现一个FilesPipeline的子类，覆写file_path方法来实现所期望的文件命名规则，这些源码文件url的最后两部分是类别和文件名，例如：

```
http://matplotlib.org/mpl_examples/(axes_grid/demo
```

可用以上括号中的部分作为文件路径，在pipelines.py实现MyFilesPipeline，代码如下：

```
from scrapy.pipelines.files import FilesPipeline
from urllib.parse import urlparse
from os.path import basename, dirname, join


class MyFilesPipeline(FilesPipeline):


    def file_path(self, request, response=None, inf
        path = urlparse(request.url).path
        return join(basename(dirname(path)), basenam
```

修改配置文件，使用MyFilesPipeline替代FilesPipeline：

```
ITEM_PIPELINES = {
```

```
        #'scrapy.pipelines.files.FilesPipeline': 1,
        'matplotlib_examples.pipelines.MyFilesPipeline':
    }
```

删除之前下载的所有文件，重新运行爬虫后，再来查看examples_src目录：

```
$ rm -r examples_src/full
$ rm examples.json
$ scrapy crawl examples -o examples.json
...
$ tree examples_src
examples_src/
├── animation
│   ├── animate_decay.py
│   ├── basic_example.py
│   ├── basic_example_writer.py
│   ├── bayes_update.py
│   ├── double_pendulum_animated.py
│   ├── dynamic_image2.py
│   ├── dynamic_image.py
│   ├── histogram.py
│   ├── moviewriter.py
│   ├── rain.py
│   ├── random_data.py
│   ├── simple_3danim.py
│   ├── simple_anim.py
│   ├── strip_chart_demo.py
│   ├── subplots.py
│   └── unchained.py
├── api
│   ├── agg_oo.py
│   ├── barchart_demo.py
│   ├── bbox_intersect.py
│   ├── collections_demo.py
│   ├── colorbar_basics.py
│   ├── colorbar_only.py
│   ├── compound_path.py
│   ├── custom_projection_examp
...省略中间输出...
│   ├── rec_edit_gtk_simple.py
│   ├── svg_histogram.py
│   ├── svg_tooltip.py
│   ├── toolmanager.py
│   └── wxcursor_demo.py
└── widgets
    ├── buttons.py
    ├── check_buttons.py
    ├── cursor.py
    ├── lasso_selector_demo.py
    ├── menu.py
    ├── multicursor.py
    ├── radio_buttons.py
    ├── rectangle_selector.py
    ├── slider_demo.py
    └── span_selector.py
26 directories, 507 files
```

从上述结果看出，507个文件按类别被下载到26个目录下，这正

是我们所期望的。

到此，文件下载的项目完成了。

9.3　项目实战：下载360图片

我们再来完成一个使用ImagesPipeline下载图片的实战项目。360图片是一个知名的图片搜索网站，在浏览器中打开http://image.so.com，页面如图9-5所示。

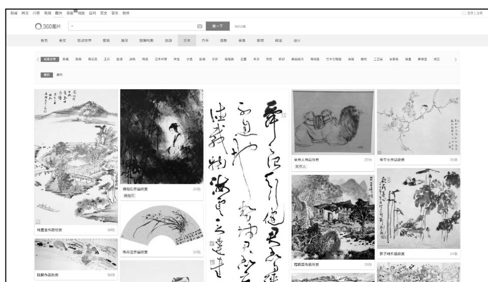


图9-5

其中，艺术分类下有大量字画图片，我们可以编写爬虫爬取这些图片。

9.3.1　项目需求

下载360图片网站中艺术分类下的所有图片到本地。

9.3.2　页面分析

在图9-5所示的页面中向下滚动鼠标滚轮，便会有更多图片加载出来，图片加载是由JavaScript脚本完成的，在图9-6中可以看到jQuery发送的请求，其响应结果是一个json串。

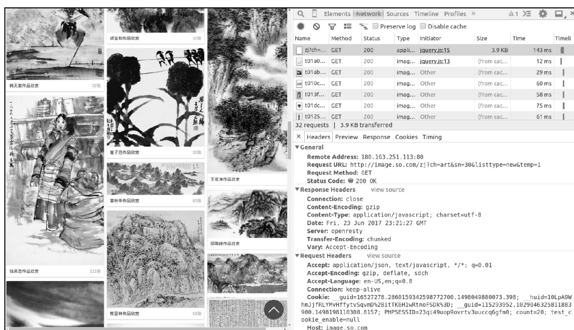


图9-6

复制图中jQuery发送请求的url，使用scrapy shell进行访问，查看响应结果的内容（json）：

```
$ scrapy shell 'http://image.so.com/zj?ch=art&sn=3
...
>>> import json
>>> res = json.loads(response.body.decode('utf8'))
>>> res
{'count': 30,
'end': False,
'lastid': 60,
'list': [{'cover_height': 942,
 'cover_imgurl': 'http://www.sinaimg.cn/dy/slidene
 'cover_width': 950,
 'dsptime': '',
 'group_title': '李正天作品欣赏',
 'grpseq': 1,
 'id': 'e4e6dbc8c5deaf2799d396569904227f',
 'imageid': '5332cbd95b1098f0e9325a16ce022a74',
 'index': 31,
 'label': '',
 'qhimg_height': 236,
 'qhimg_thumb_url': 'http://p0.so.qhimgs1.com/sdr/
 'qhimg_url': 'http://p0.so.qhimgs1.com/t01ab50e7f
 'qhimg_width': 238,
 'tag': '新浪艺术名家人物库',
```

```
 'total_count': 70},
 {'cover_height': 1798,
 'cover_imgurl': 'http://www.sinaimg.cn/dy/slidene
 'cover_width': 950,
 'dsptime': '',
 'group_title': '崔自默作品欣赏',
 'grpseq': 1,
 'id': 'f08148a113c6c2e6104a77798d285d88',
 'imageid': 'c6662a238bb6faf9b22a335db6707fff',
 'index': 32,
 'label': '',
 'qhimg_height': 450,
 'qhimg_thumb_url': 'http://p0.so.qhmsg.com/sdr/23
 'qhimg_url': 'http://p0.so.qhmsg.com/t01b187fc2ce
 'qhimg_width': 238,
 'tag': '新浪艺术名家人物库',
 'total_count': 53},
 {'cover_height': 950,
 'cover_imgurl': 'http://www.sinaimg.cn/dy/slidene
 'cover_width': 950,
 'dsptime': '',
'group_title': '徐宁作品欣赏',
'grpseq': 1,
'id': 'ed8686ac7f10dfb52d68baca348a08be',
'imageid': '51c2b804fb6d402486737c29c5301a84',
'index': 33,
'label': '',
'qhimg_height': 238,
'qhimg_thumb_url': 'http://p2.so.qhmsg.com/sdr/238
'qhimg_url': 'http://p2.so.qhmsg.com/t017f259639fd
'qhimg_width': 238,
'tag': '新浪艺术名家人物库',
'total_count': 161},


...省略中间部分...


{'cover_height': 377,
'cover_imgurl': 'http://www.sinaimg.cn/dy/slidenew
```

```
    'cover_width': 950,
    'dsptime': '',
    'group_title': '王国斌作品欣赏',
    'grpseq': 1,
    'id': '8e173e45250d90d2dc7316777e2be59b',
    'imageid': 'c7d7e74dc18685f5c100d235522d5e4b',
    'index': 59,
    'label': '',
    'qhimg_height': 94,
    'qhimg_thumb_url': 'http://p2.so.qhimgs1.com/sdr/2
    'qhimg_url': 'http://p2.so.qhimgs1.com/t014d248b01
    'qhimg_width': 238,
    'tag': '新浪艺术名家人物库',
    'total_count': 13},
    {'cover_height': 1215,
    'cover_imgurl': 'http://www.sinaimg.cn/dy/slidenew
    'cover_width': 900,
    'dsptime': '',
    'group_title': '王习三作品欣赏',
    'grpseq': 1,
    'id': '989031bb929f667f8eb920cfa21e32fa',
    'imageid': 'f57b9882a93265edcd85e59d3fbb8a4c',
    'index': 60,
    'label': '王习三',
    'qhimg_height': 321,
    'qhimg_thumb_url': 'http://p4.so.qhmsg.com/sdr/238
    'qhimg_url': 'http://p4.so.qhmsg.com/t015381735d7c
    'qhimg_width': 238,
    'tag': '新浪艺术名家人物库',
    'total_count': 31}]}
```


如上所示，响应结果（json）中的list字段是一个图片信息列表，count字段是列表中图片信息的数量，每一项图片信息的qhimg_url字段是图片下载地址。

连续滚动鼠标滚轮加载图片，捕获更多jQuery发送的请求：

第1次加载：http://image.so.com/zj?ch=art&sn=30&listtype=new&temp=1

第2次加载：http://image.so.com/zj?ch=art&sn=60&listtype=new&temp=1

第3次加载：http://image.so.com/zj?ch=art&sn=90&listtype=new&temp=1

……

经过观察，可以总结出这些url的规律：

- ch参数　分类标签。
- sn参数　从第几张图片开始加载，即结果列表中第一张图片在服务器端的序号。

我们可以通过这个API每次获取固定数量的图片信息，从中提取每一张图片的url（下载地址），直到响应结果中的count字段为0（意味着没有更多图片了）。

到此，页面分析工作完成了。

9.3.3　编码实现

接下来，我们按以下3步完成该项目：

（1）创建Scrapy项目，并使用scrapy genspider命令创建Spider。

（2）在配置文件中启用ImagesPipeline，并指定图片下载目录。

（3）实现ImagesSpider。

步骤　**01**　首先创建Scrapy项目，取名为so_image，再使用scrapy genspider命令创建Spider。

```
$ scrapy startproject so_image
$ cd so_image
$ scrapy genspider images image.so.com
```

步骤　**02**　在配置文件settings.py中启用ImagesPipeline，并指定图片下载目录，代码如下：

```
ITEM_PIPELINES = {
  'scrapy.pipelines.images.ImagesPipeline': 1,
}
IMAGES_STORE = 'download_images'
```

步骤　**03**　实现IamgesSpider，代码如下：

```
    # -*- coding: utf-8 -*-
    import scrapy
    from scrapy import Request
    import json


    class ImagesSpider(scrapy.Spider):
        BASE_URL = 'http://image.so.com/zj?ch=art&
        start_index = 0


        # 限制最大下载数量，防止磁盘用量过大
        MAX_DOWNLOAD_NUM = 1000


        name = "images"
        start_urls = [BASE_URL % 0]


        def parse(self, response):
            # 使用json 模块解析响应结果
            infos = json.loads(response.body.decode
            # 提取所有图片下载url 到一个列表， 赋给item的
            yield {'image_urls': [info['qhimg_url']]
```

```
            # 如count 字段大于0，并且下载数量不足MAX_DOW
图片信息
            self.start_index += infos['count']
            if infos['count'] > 0 and self.start_in
               yield Request(self.BASE_URL % self.s
```

编码完成后，运行爬虫：

```
$ scrapy crawl images
```

运行结束后，查看图片下载目录download_images，如图9-7所示，我们成功爬取到了607张艺术图片。

图9-7

到此，图片下载的项目完成了。

9.4 本章小结

本章我们学习了在Scrapy中下载文件和图片的方法，先简单介绍了Scrapy提供的FilesPipeline和ImagesPipeline，然后通过两个实战项目演示了FilesPipeline和ImagesPipeline的使用。

第10章
模拟登录

目前，大部分网站都具有用户登录功能，其中某些网站只有在用户登录后才能获得有价值的信息，在爬取这类网站时，Scrapy爬虫程序需要先模拟用户登录，再爬取内容，这一章来学习在Scrapy中模拟登录的方法。

10.1　登录实质

在学习模拟登录前，应先对网站登录的原理有所了解，我们在Chrome浏览器中跟踪一次实际的登录操作，观察浏览器与网站服务器是如何交互的。以登录http://example.webscraping.com网站为例进行演示，这是一个专门用于练习爬虫技术的网站，图10-1所示为该网站的登录页面。

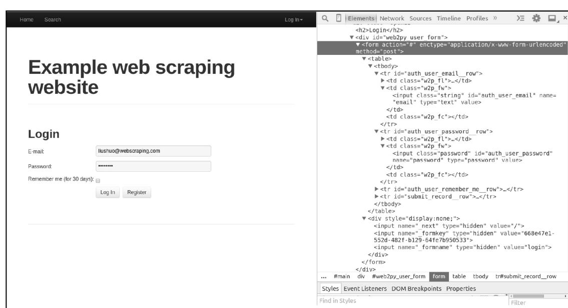


图10-1

页面中的表单对应于HTML中的＜form＞元素，当填写完表单，单击“提交”按钮时，浏览器会根据＜form＞元素的内容发送一个HTTP请求给服务器，其中：

- ＜form＞的method属性决定了HTTP请求的方法（本例中为POST）。

- ＜form＞的action属性决定了HTTP请求的url（本例中为#，也就是当前页面的url）。
- ＜form＞的enctype属性决定了表单数据的编码类型（本例中为x-www-urlencoded）。
- ＜form＞中的＜input＞元素决定了表单数据的内容。

再来看＜form＞中的＜input＞元素：

- name属性为'email'和'password'的两个＜input＞，对应于账号和密码输入框，它们的值待用户填写。
- 在＜div style="display:none;"＞中还包含了3个隐藏的＜input type="hidden"＞，它们的值在value属性中，虽然值不需要用户填写，但提交的表单数据中缺少它们可能会导致登录验证失败，这些隐藏的＜input＞有其他一些用途，比如：
  - ＜input name="_next"＞用来告诉服务器，登录成功后页面跳转的地址。
  - ＜input name="_formkey"＞用来防止CSRF跨域攻击，相关内容请查阅资料。

接下来，填入账号密码，在Chrome开发者工具中观察单击Log In按钮后浏览器发送的HTTP请求，如图10-2所示。

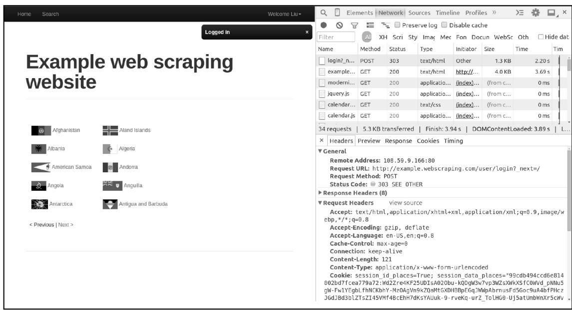


图10-2

从图10-2中可以看出，我们捕获到了很多HTTP请求，其中第一

个就是发送登录表单的POST请求，查看该HTTP请求，可以找到请求url、请求方法（method）、请求头部（headers）、表单数据（form data）等信息，其中我们最关心的是表单数据，如图10-3所示。

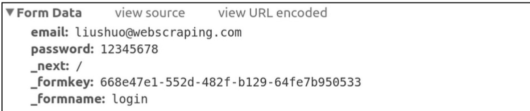


图10-3

表单数据由多个键值对构成，每个键值对对应一个＜input＞元素，其中的键是＜input＞的name属性，值是＜input＞的value属性（用户填写的内容会成为＜input＞的value）。需要注意的是，图10-3中显示的并不是实际的POST正文（content），这样的显示方式只是为了方便用户查看。单击view source按钮，可以看到实际的POST正文，如图10-4所示。

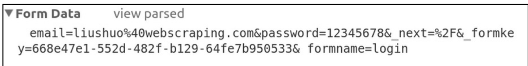


图10-4

分析过HTTP请求信息后，再来看HTTP响应信息，如图10-5所示。

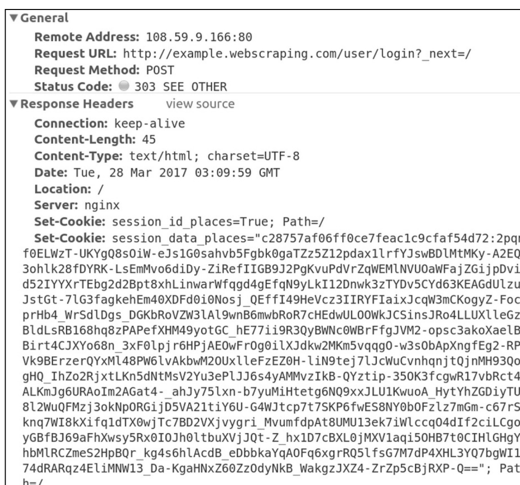


图10-5

响应头部中长长的Set-Cookie字段就是网站服务器程序保存在客户端（浏览器）的Cookie信息，其中包含标识用户身份的session信息，之后对该网站发送的其他HTTP请求都会带上这个“身份证”（session信息），服务器程序通过这个“身份证”识别出发送请求的用户，从而决定响应怎样的页面。另外，响应的状态码是303，它代表页面重定向，浏览器会读取响应头部中的Location字段，依据其中描述的路径（本例中为／）再次发送一个GET请求，图10-6所示为这个GET请求的信息。

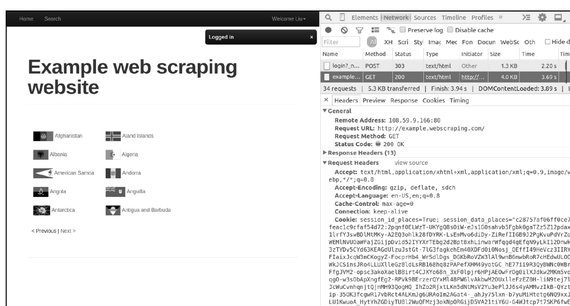


图10-6

观察请求头部中的Cookie字段，它携带了之前POST请求获取的

Cookie信息，最终浏览器用该请求响应的HTML文档刷新了页面，在页面右上角可以看到Welcome　Liu的字样，表明这是登录成功后的页面。

10.2　Scrapy模拟登录

10.2.1　使用FormRequest

现在大家了解了登录的实质，其核心是向服务器发送含有登录表单数据的HTTP请求（通常是POST）。Scrapy提供了一个FormRequest类（Request的子类），专门用于构造含有表单数据的请求，FormRequest的构造器方法有一个formdata参数，接收字典形式的表单数据。接下来，我们在scrapy　　　　shell环境下演示如何使用FormRequest模拟登录。

首先爬取登录页面http://example.webscraping.com/user/login：

```
$ scrapy shell http://example.webscraping.com/user
```

通过之前的分析，我们已经了解了表单数据应包含的信息：账号和密码信息，再加3个隐藏<input>中的信息。先把这些信息收集到一个字典中，然后使用这个表单数据字典构造FormRequest对象：

```
>>> # 先提取3 个隐藏<input>中包含的信息，它们在<div sty
>>> sel = response.xpath('//div[@style]/input')
>>> sel
[<Selector xpath='//div[@style]/input' data='<inpu
 <Selector xpath='//div[@style]/input' data='<inpu
 <Selector xpath='//div[@style]/input' data='<inpu
>>> # 构造表单数据字典
>>> fd = dict(zip(sel.xpath('./@name').extract(),
>>> fd
```

```
{'_formkey': '432dcb0c-0d85-443f-bb50-9644cfeb2f2b
 '_formname': 'login',
 '_next': '/'}
>>> # 填写账号和密码信息
>>> fd['email'] = 'liushuo@webscraping.com'
>>> fd['password'] = '12345678'
>>> fd
{'_formkey': '432dcb0c-0d85-443f-bb50-9644cfeb2f2b
 '_formname': 'login',
 '_next': '/',
 'email': 'liushuo@webscraping.com',
 'password': '12345678'}
>>> from scrapy.http import FormRequest
>>> request = FormRequest('http://example.webscrap
```

以上是直接构造FormRequest对象的方式，除此之外还有一种更为简单的方式，即调用FormRequest的from_response方法。调用时需传入一个Response对象作为第一个参数，该方法会解析Response对象所包含页面中的<form>元素，帮助用户创建FormRequest对象，并将隐藏<input>中的信息自动填入表单数据。使用这种方式，我们只需通过formdata参数填写账号和密码即可，代码如下：

```
>>> fd = {'email': 'liushuo@webscraping.com', 'pas
>>> request = FormRequest.from_response(response,
```

使用任意方式构造好FormRequest对象后，接下来提交表单请求：

```
>>> fetch(request)
[scrapy] DEBUG: Redirecting (303) to <GET http://e
<POST http://example.webscraping.com/user/login>
[scrapy] DEBUG: Crawled (200) <GET http://example.
```

在log信息中，可以看到和浏览器提交表单时类似的情形：POST

请求的响应状态码为303，之后Scrapy自动再发送一个GET请求下载跳转页面。此时，可以通过在页面中查找特殊字符串或在浏览器中查看页面验证登录是否成功，如图10-7所示。

```
>>> 'Welcome Liu' in response.text
True
>>> view(response)
```

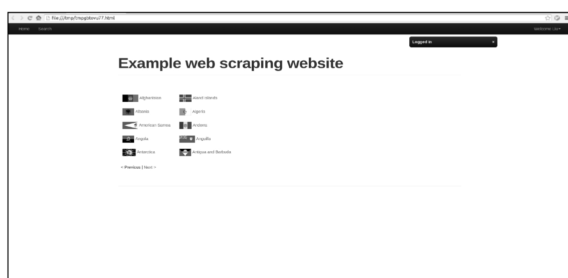


图10-7

验证结果表明模拟登录成功了。显然，Scrapy发送的第2个GET请求携带了第1个POST请求获取的Cookie信息，为请求附加Cookie信息的工作是由Scrapy内置的下载中间件CookiesMiddleware自动完成的。现在，我们可以继续发送请求，爬取那些只有登录后才能获取的信息了，这里以爬取用户个人信息为例，如图10-8所示。

```
>>> fetch('http://example.webscraping.com/user/pro
[scrapy] DEBUG: Crawled (200) <GET http://example.
>>> view(response)
```

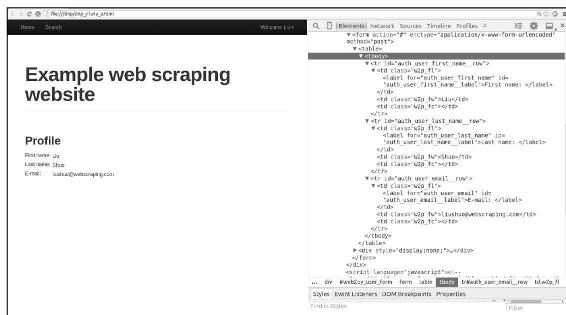


图10-8

提取页面中的用户个人信息，代码如下：

```
>>> keys = response.css('table label::text').re('(
>>> keys
['First name', 'Last name', 'E-mail']
>>> values = response.css('table td.w2p_fw::text')
>>> values
['Liu', 'Shuo', 'liushuo@webscraping.com']
>>> dict(zip(keys, values))
{'E-mail': 'liushuo@webscraping.com', 'First name'
```

到此，使用FormRequest模拟登录的过程就演示完了。

10.2.2　实现登录Spider

整理10.2.1小节中登录http://example.webscraping.com后爬取用户个人信息的代码，实现一个LoginSpider，代码如下：

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy.http import Request, FormRequest
class LoginSpider(scrapy.Spider):
    name = "login"
    allowed_domains = ["example.webscraping.com"]
```

```
    start_urls = ['http://example.webscraping.com/u

    def parse(self, response):
        # 解析登录后下载的页面，此例中为用户个人信息页面
        keys = response.css('table label::text').re
        values = response.css('table td.w2p_fw::tex

        yield dict(zip(keys, values))

    # ----------------------------登录--------------
    # 登录页面的url
    login_url = 'http://example.webscraping.com/use

    def start_requests(self):
        yield Request(self.login_url, callback=self

    def login(self, response):
        # 登录页面的解析函数，构造FormRequest对象提交表单
        fd = {'email': 'liushuo@webscraping.com', '
        yield FormRequest.from_response(response, f
                                        callback=self.

    def parse_login(self, response):
        # 登录成功后，继续爬取start_urls 中的页面
        if 'Welcome Liu' in response.text:
            yield from super().start_requests() # P
```

解释上述代码如下：

- 覆写基类的start_requests方法，最先请求登录页面。
- login方法为登录页面的解析函数，在该方法中进行模拟登录，构造表单请求并提交。

- parse_login方法为表单请求的响应处理函数，在该方法中通过在页面查找特殊字符串'Welcome Liu'判断是否登录成功，如果成功，调用基类的start_requests方法，继续爬取start_urls中的页面。

我们这样设计LoginSpider就是想把模拟登录和爬取内容的代码分离开，使得逻辑上更加清晰。

10.3 识别验证码

目前，很多网站为了防止爬虫爬取，登录时需要用户输入验证码。下面我们学习如何在爬虫程序中识别验证码（在举例过程中不指明具体网站）。

图10-9所示为某网站登录页面，其中包含验证码。

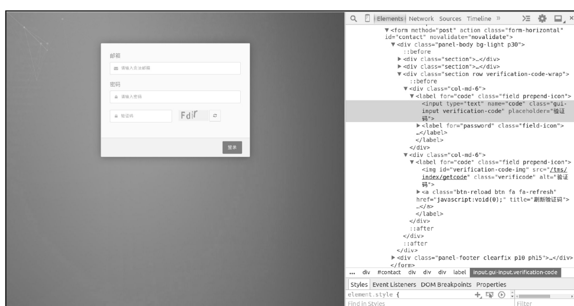


图10-9

页面中的验证码图片对应一个＜img＞元素，即一张图片，浏览器加载完登录页面后，会携带之前访问获取的Cookie信息，继续发送一个HTTP请求加载验证码图片。和账号密码输入框一样，验证码输入框也对应一个＜input＞元素，因此用户输入的验证码会成为表单数据的一部分，表单提交后由网站服务器程序验证。

识别验证码有多种方式，下面介绍常用的几种。

10.3.1 OCR识别

OCR是光学字符识别的缩写，用于在图像中提取文本信息，tesseract-ocr是利用该技术实现的一个验证码识别库，在Python中可以通过第三方库pytesseract调用它。下面介绍如何使用pytesseract识别验证码。

首先安装tesseract-ocr，在Ubuntu下可以使用apt-get安装：

```
sudo apt-get install tesseract-ocr
```

接下来安装pytesseract，它依赖于Python图像处理库PIL或Pillow（PIL和Pillow功能类似，任选其一），可以使用pip安装它们：

```
$ pip install pillow
$ pip install pytesseract
```

现在，我们使用pytesseract识别图片code.png中的验证码，代码如下：




```
>>> from PIL import Image
>>> import pytesseract
>>> img = Image.open('code.png')
>>> img = img.convert('L')
>>> pytesseract.image_to_string(img)
'cqKE'
```

上面的代码中，先使用Image.open打开图片，为了提高识别率，调用Image对象的convert方法把图片转换为黑白图，然后将黑白图传递给pytesseract.image_to_string方法进行识别，这里我们幸运地识别

成功了。经测试，此段代码对于X网站中的验证码识别率可以达到72%，这已经足够高了。

下面我们以之前的LoginSpider为模板实现一个使用pytesseract识别验证码登录的Spider：

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy import Request, FormRequest
import json
from PIL import Image
from io import BytesIO
import pytesseract
from scrapy.log import logger


class CaptchaLoginSpider(scrapy.Spider):
name = "login_captcha"
start_urls = ['http://XXX.com/']


def parse(self, response):
    ...
# X 网站登录页面的url（虚构的）
login_url = 'http://XXX.com/login'
user      = 'liushuo@XXX.com'
password  = '12345678'


def start_requests(self):
    yield Request(self.login_url, callback=self.log
def login(self, response):
    # 该方法既是登录页面的解析函数，又是下载验证码图片的响应
    # 如果response.meta['login_response']存在，当前re
    # 否则当前response 为登录页面的响应
    login_response = response.meta.get('login_respo
```

```
    if not login_response:
        # Step 1:
        # 此时response 为登录页面的响应，从中提取验证码图
        captchaUrl = response.css('label.field.prep
                                .extract_first()


        captchaUrl = response.urljoin(captchaUrl)
        # 构造Request 时，将当前response 保存到meta 字
        yield Request(captchaUrl,
                      callback=self.login,
                      meta={'login_response': response
                      dont_filter=True)


    else:
        # Step 2:
        # 此时，response 为验证码图片的响应，response.b
        # login_response 为登录页面的响应，用其构造表单i
        formdata = {
           'email': self.user,
           'pass': self.password,
           # 使用OCR识别
           'code': self.get_captcha_by_OCR(response
        }
        yield FormRequest.from_response(login_respo
                                        callback=se
                                        formdata=fo


def parse_login(self, response):
    # 根据响应结果判断是否登录成功
    info = json.loads(response.text)
    if info['error'] == '0':
        logger.info('登录成功:-)')
        return super().start_requests()
    logger.info('登录失败:-(, 重新登录...')
    return self.start_requests()
def get_captcha_by_OCR(self, data):
    # OCR识别
```

```
    img = Image.open(BytesIO(data))
    img = img.convert('L')
    captcha = pytesseract.image_to_string(img)
    img.close()
    return captcha
```

解释上述代码如下：

- login方法

  带有验证码的登录，需要额外发送一个HTTP请求来获取验证码图片，这里的login方法既处理下载登录页面的响应，又处理下载验证码图片的响应。

  - 解析登录页面时，提取验证码图片的url，发送请求下载图片，并将登录页面的Response对象保存到Request对象的meta字典中。
  - 处理下载验证码图片的响应时，调用get_captcha_by_OCR方法识别图片中的验证码，然后将之前保存的登录页面的Response对象取出，构造FormRequest对象并提交。

- get_captcha_by_OCR方法

  参数data是验证码图片的二进制数据，类型为bytes，想要使用Image.open函数构造Image对象，先要把图片的二进制数据转换成某种类文件对象，这里使用BytesIO进行包裹，获得Image对象后先将其转换成黑白图，然后调用pytesseract.image_to_string方法进行识别。

- parse_login方法

  处理表单请求的响应。响应正文是一个json串，其中包含了用户验证的结果，先调用json.loads将正文转换为Python字典，然后依据其中error字段的值判断登录是否成功，若登录成功，则从start_urls中的页面开始爬取；若登录失败，则重新登录。

10.3.2 网络平台识别

在10.3.1小节中，使用pytesseract识别的验证码比较简单，对于某些复杂的验证码，pytesseract的识别率很低或者无法识别。目前，有很多网站专门提供验证码识别服务，可以识别较为复杂的验证码（有些是人工处理的），它们被称之为验证码识别平台，这些平台多数是付费使用的，价格大约为1元钱识别100个验证码，平台提供了HTTP服务接口，用户可以通过HTTP请求将验证码图片发送给平台，平台识别后将结果通过HTTP响应返回。

在阿里云市场可以找到很多验证码识别平台，我们随意挑选了一个，如图10-10和图10-11所示。

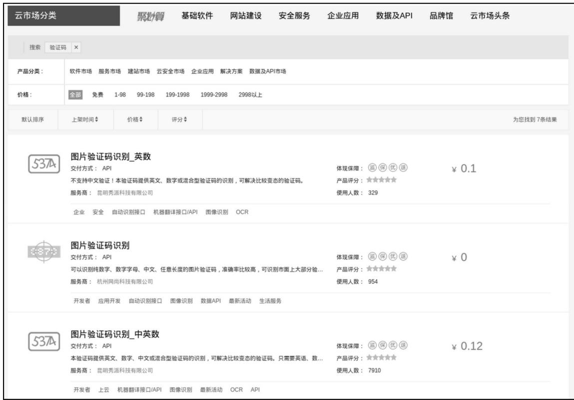


图10-10

图片验证码识别_中英数

调用地址：http://ali-checkcode.showapi.com/checkcode

请求方式：POST

图片验证码识别_中英数

调用地址：http://ali-checkcode.showapi.com/checkcode

请求方式：POST

返回类型：JSON

API 调用：API 简单身份认证调用方法（APPCODE）展开

调试工具：去调试

请求参数（Headers）

请求参数（Query）

请求参数（Body）

名称	类型	是否必须	描述
convert_to_jpg	STRING	可选	有少量png或gif图转成jpg格式后识别率明显提高（并不是所有png或gif转成jpg后都会提高识别率）。 此字段为1时表示需要把图片转为jpg格式，其他值不做转换操作。
img_base64	STRING	必选	图片文件的base64字符串。图片大小需要小于100KB。
typeId	STRING	必选	1.纯数字 typeId=1000 任意长度数字，识别率会降低 typeId=1010 1位数字 typeId=1020 2位数字 ... typeId=1100 10位数字 2.纯英文 typeId=2000 任意长度英文，识别率会降低 typeId=2010 1位英文 typeId=2020 2位英文 ... typeId=2100 10位英文 3.英文数字混合 typeId=3000 任意长度英数混合，识别率会降低 typeId=3010 1位英数 typeId=3020 2位英数混合 ... typeId=3100 10位英数混合 4.纯汉字 typeId=4000 任意长度汉字混合，识别率会降低 typeId=4010 1位汉字 typeId=4020 2位汉字 ... typeId=4100 10位

图10-11

购买服务后，我们利用该平台识别图片code.gif中较为复杂的验证码：



阅读API文档，实现代码如下：

```
import requests
import base64
from pprint import pprint


# 购买服务后，平台发放给我们一个appcode，用来识别请求者的身
APPCODE = 'f239ccawf37f287418a90e2f922649273c4'
```

```
url = 'http://ali-checkcode.showapi.com/checkcode'
img_data = open('code.gif', 'rb').read()
form = {}


# 不转换为jpg
form['convert_to_jpg'] = '0'
# 对图片进行base64 编码
form['img_base64'] = base64.b64encode(img_data)
# 7 位汉字
form['typeId'] = '4070'


# 用户验证
headers = {'Authorization': 'APPCODE ' + APPCODE}
response = requests.post(url, headers=headers, dat
pprint(response.json())
```

表单中3个字段含义如下：

- convert_to_jpg是否将图片转换为jpg格式。
- img_base64图片数据的base64编码。
- typeId验证码类型，这里的'4070'代表7位汉字。

运行脚本，并观察结果：

```
$ python3 ali_checkcode.py
{'showapi_res_body': {'Id': '4d5fea-21eb-4043-8236
                      'Result': '损俱饶现渊弹翠',
                      'ret_code': 0},
 'showapi_res_code': 0,
 'showapi_res_error': ''}
```

如上所示，识别成功了。

现在，我们把使用平台识别验证码的方式也加入

CaptchaLoginSpider中，只需添加一个get_captcha_by_network方法：

```
class CaptchaLoginSpider(scrapy.Spider):
    ...


    def get_captcha_by_OCR(self, data):
        # OCR识别
        img = Image.open(BytesIO(data))
        img = img.convert('L')
        captcha = pytesseract.image_to_string(img)
        img.close()


        return captcha


    def get_captcha_by_network(self, data):
        # 平台识别
        import requests


        url = 'http://ali-checkcode.showapi.com/chec
        appcode = 'f23cca37f287418a90e2f922649273c4'


        form = {}
        form['convert_to_jpg'] = '0'
        form['img_base64'] = base64.b64encode(data)
        form['typeId'] = '3040'


        headers = {'Authorization': 'APPCODE ' + app
        response = requests.post(url, headers=header
        res = response.json()
```

```
        if res['showapi_res_code'] == 0:
            return res['showapi_res_body']['Result']

        return ''
```

10.3.3 人工识别

最后讲解的方法听起来似乎很笨：人工识别。通常网站只需登录一次便可爬取，在其他识别方式不管用时，人工识别一次验证码也是可行的，其实现也非常简单——在Scrapy下载完验证码图片后，调用Image.show方法将图片显示出来，然后调用Python内置的input函数，等待用户肉眼识别后输入识别结果。

我们将人工识别的方式也加入CaptchaLoginSpider中，再添加一个get_captcha_by_user方法：

```
class CaptchaLoginSpider(scrapy.Spider):
    ...

    def get_captcha_by_OCR(self, data):
        # OCR识别
        img = Image.open(BytesIO(data))
        img = img.convert('L')
        captcha = pytesseract.image_to_string(img)
        img.close()

        return captcha

    def get_captcha_by_network(self, data):
        # 平台识别
        import requests
```

```
        url = 'http://ali-checkcode.showapi.com/che
        appcode = 'f23cca37f287418a90e2f922649273c4

        form = {}
        form['convert_to_jpg'] = '0'
        form['img_base64'] = base64.b64encode(data)
    form['typeId'] = '3040'

    headers = {'Authorization': 'APPCODE ' + appcod
    response = requests.post(url, headers=headers,
    res = response.json()

    if res['showapi_res_code'] == 0:
        return res['showapi_res_body']['Result']

    return ''

def get_captcha_by_user(self, data):
    # 人工识别
    img = Image.open(BytesIO(data))
    img.show()
    captcha = input('输入验证码:')
    img.close()
    return captcha
```

10.4　Cookie登录

在10.3节，我们讲解了识别验证码登录的方法，但目前网站的验证码越来越复杂，某些验证码已经复杂到人类难以识别的程度，有些时候提交表单登录的路子难以走通。此时，我们可以换一种登录爬取的思路，在使用浏览器登录网站后，包含用户身份信息的Cookie会被浏览器保存在本地，如果Scrapy爬虫能直接使用浏览器中的Cookie发

送HTTP请求，就可以绕过提交表单登录的步骤。

10.4.1　获取浏览器Cookie

我们无须费心钻研，各种浏览器将Cookie以哪种形式存储在哪里，使用第三方Python库browsercookie便可以获取Chrome和Firefox浏览器中的Cookie。

使用pip安装browsercookie：

```
pip install browsercookie
```

browsercookie的使用非常简单，示例代码如下：

```
>>> import browsercookie
>>> chrome_cookiejar = browsercookie.chrome()
>>> firefox_cookiejar = browsercookie.firefox()   #
>>> type(chrome_cookiejar)
http.cookiejar.CookieJar
>>> for cookie in chrome_cookiejar:
...     print(cookie)
```

browsercookie的chrome和firefox方法分别返回Chrome和Firefox浏览器中的Cookie，返回值是一个http.cookiejar.CookieJar对象，对CookieJar对象进行迭代，可以访问其中的每个Cookie对象。

10.4.2　CookiesMiddleware源码分析

之前曾提到过，Scrapy爬虫所使用的Cookie由内置下载中间件CookiesMiddleware自动处理。下面我们来分析一下CookiesMiddleware是如何工作的，其源码如下：

```
import os
import six
import logging
from collections import defaultdict


from scrapy.exceptions import NotConfigured
from scrapy.http import Response
from scrapy.http.cookies import CookieJar
from scrapy.utils.python import to_native_str


logger = logging.getLogger(__name__)


class CookiesMiddleware(object):
   """This middleware enables working with sites t


   def __init__(self, debug=False):
      self.jars = defaultdict(CookieJar)
      self.debug = debug


   @classmethod
   def from_crawler(cls, crawler):
      if not crawler.settings.getbool('COOKIES_ENA
        raise NotConfigured
   return cls(crawler.settings.getbool('COOKIES_DE


def process_request(self, request, spider):
   if request.meta.get('dont_merge_cookies', False
        return


   cookiejarkey = request.meta.get("cookiejar")
   jar = self.jars[cookiejarkey]
   cookies = self._get_request_cookies(jar, reques
   for cookie in cookies:
```

```
            jar.set_cookie_if_ok(cookie, request)


        # set Cookie header
        request.headers.pop('Cookie', None)
        jar.add_cookie_header(request)
        self._debug_cookie(request, spider)


    def process_response(self, request, response, spid
        if request.meta.get('dont_merge_cookies', False
            return response


        # extract cookies from Set-Cookie and drop inva
        cookiejarkey = request.meta.get("cookiejar")
        jar = self.jars[cookiejarkey]
        jar.extract_cookies(response, request)
        self._debug_set_cookie(response, spider)


        return response


    def _debug_cookie(self, request, spider):
        if self.debug:
            cl = [to_native_str(c, errors='replace')
                  for c in request.headers.getlist('Cook
            if cl:
                cookies = "\n".join("Cookie: {}\n".forma
                msg = "Sending cookies to: {}\n{}".forma
                logger.debug(msg, extra={'spider': spide
    def _debug_set_cookie(self, response, spider):
        if self.debug:
            cl = [to_native_str(c, errors='replace')
                  for c in response.headers.getlist('Set
            if cl:
                cookies = "\n".join("Set-Cookie: {}\n".f
                msg = "Received cookies from: {}\n{}".fo
                logger.debug(msg, extra={'spider': spide
```

```
    def _format_cookie(self, cookie):
        # build cookie string
        cookie_str = '%s=%s' % (cookie['name'], cookie[


        if cookie.get('path', None):
            cookie_str += '; Path=%s' % cookie['path']
        if cookie.get('domain', None):
            cookie_str += '; Domain=%s' % cookie['domai


        return cookie_str


    def _get_request_cookies(self, jar, request):
        if isinstance(request.cookies, dict):
            cookie_list = [{'name': k, 'value': v} for
                        six.iteritems(request.cookies)]
        else:
            cookie_list = request.cookies


        cookies = [self._format_cookie(x) for x in cook
        headers = {'Set-Cookie': cookies}
        response = Response(request.url, headers=header


        return jar.make_cookies(response, request)
```

分析其中几个核心方法如下：

- from_crawler方法

  从配置文件中读取COOKIES_ENABLED，决定是否启用该中间件。如果启用，调用构造器创建对象，否则抛出NotConfigured异常，Scrapy将忽略该中间件。

- _init_方法

使用标准库中的collections.defaultdict创建一个默认字典self.jars，该字典中每一项的值都是一个scrapy.http.cookies.CookieJar对象，CookiesMiddleware可以让Scrapy爬虫同时使用多个不同的CookieJar。例如，在某网站我们注册了两个账号account1和account2，假设一个爬虫想同时登录两个账号对网站进行爬取，为了避免Cookie冲突（通俗地讲，登录一个会替换掉另一个），此时可以让每个账号发送的HTTP请求使用不同的CookieJar，在构造Request对象时，可以通过meta参数的cookiejar字段指定所要使用的CookieJar，如：

```
# 账号account1 发送的请求
Request(url1, meta={'cookiejar': 'account1'})
Request(url2, meta={'cookiejar': 'account1'})
Request(url3, meta={'cookiejar': 'account1'})
...
# 账号account2 发送的请求
Request(url1, meta={'cookiejar': 'account2'})
Request(url2, meta={'cookiejar': 'account2'})
Request(url3, meta={'cookiejar': 'account2'})
...
```

- process_request方法

  处理每一个待发送的Request对象，尝试从request.meta['cookiejar']获取用户指定使用的CookieJar，如果用户未指定，就使用默认的CookieJar(self.jars[None])。调用self._get_request_cookies方法获取发送请求request应携带的Cookie信息，填写到HTTP请求头部。

- process_response方法

  处理每一个Response对象，依然通过request.meta['cookiejar']获取CookieJar对象，调用

extract_cookies方法将HTTP响应头部中的Cookie信息保存到CookieJar对象中。

另外需要注意的是，这里的CookieJar是scrapy.http.cookies.CookieJar，而10.4.1小节中的CookieJar是标准库中的http.cookiejar.CookieJar，它们是不同的类，前者对后者进行了包装，两者可以相互转化。

10.4.3 实现BrowserCookiesMiddleware

CookiesMiddleware自动处理Cookie的特性给用户提供了便利，但它不能使用浏览器的Cookie，我们可以利用browsercookie对CookiesMiddleware进行改良，实现一个能使用浏览器Cookie的中间件，代码如下：

```
import browsercookie
from scrapy.downloadermiddlewares.cookies import C


class BrowserCookiesMiddleware(CookiesMiddleware):
   def __init__(self, debug=False):
       super().__init__(debug)
       self.load_browser_cookies()


   def load_browser_cookies(self):
       # 加载Chrome 浏览器中的Cookie
       jar = self.jars['chrome']
       chrome_cookiejar = browsercookie.chrome()
       for cookie in chrome_cookiejar:
           jar.set_cookie(cookie)


       # 加载Firefox 浏览器中的Cookie
       jar = self.jars['firefox']
       firefox_cookiejar = browsercookie.firefox()
```

```
        for cookie in firefox_cookiejar:
            jar.set_cookie(cookie)
```

了解了CookiesMiddleware的工作原理，便不难理解BrowserCookiesMiddleware的实现了，其核心思想是：在构造BrowserCookiesMiddleware对象时，使用browsercookie将浏览器中的Cookie提取，存储到CookieJar字典self.jars中，解释代码如下：

- 继承CookiesMiddleware并实现构造器方法，在构造器方法中先调用基类的构造器方法，然后调用self.load_browser_cookies方法加载浏览器Cookie。
- 在load_browser_cookies方法中，使用self.jars['chrome']和self.jars['firefox']从默认字典中获得两个CookieJar对象，然后调用browsercookie的chrome和firefox方法，分别获取两个浏览器中的Cookie，将它们填入各自的CookieJar对象中。

10.4.4 爬取知乎个人信息

下面通过一个例子展示BrowserCookiesMiddleware的使用，知乎网是国内最流行的问答网站，我们在Chrome浏览器登录知乎后，可以访问如图10-12所示的用户个人信息页面。

接下来，我们使用BrowserCookiesMiddleware爬取这个登录后才能访问的页面，提取用户的“姓名”和“个性域名”信息。



图10-12

首先创建Scrapy项目，取名为browser_cookie：

```
$ scrapy startproject browser_cookie
```

然后，将BrowserCookiesMiddleware源码复制到该项目下的middlewares.py中，并在配置文件settings.py中添加如下配置：

```
# 伪装成常规浏览器
USER_AGENT = 'Mozilla/5.0 (X11; Linux x86_64) Chrc
# 用BrowserCookiesMiddleware 替代CookiesMiddleware
DOWNLOADER_MIDDLEWARES = {
    'scrapy.downloadermiddlewares.cookies.CookiesMi
    'browser_cookie.middlewares.BrowserCookiesMiddl
}
```

由于需求非常简单，因此不再编写Spider，直接在scrapy shell环境中进行演示。注意，为了使用项目中的配置，需要在项目目录下启动scrapy shell命令：

```
$ scrapy shell
...
>>> from scrapy import Request
>>> url = 'https://www.zhihu.com/settings/profile'
>>> fetch(Request(url, meta={'cookiejar': 'chrome'
...
>>> view(response)
```

调用view函数后，在浏览器中可看到如图10-13所示的页面。

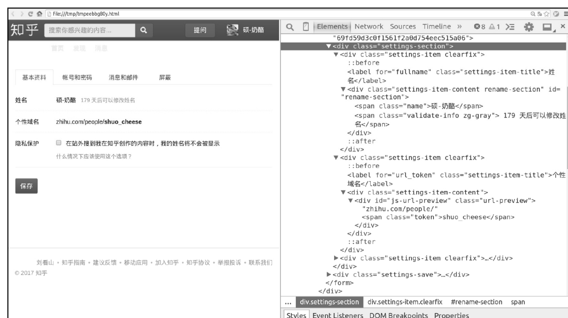


图10-13

结果表明，BrowserCookiesMiddleware按我们所预期的工作了，Scrapy爬虫使用浏览器的Cookie成功地获取了一个需要用户登录后才能访问的页面。

最后，提取页面中的“姓名”和“个性域名”信息：

```
>>> response.css('div#rename-section span.name::te
'硕-奶酪'
>>> response.xpath('string(//div[@id="js-url-previ
'zhihu.com/people/shuo_cheese'
```

到此，使用浏览器Cookie登录的案例展示完毕。

10.5　本章小结

本章我们学习了Scrapy爬虫模拟登录网站的相关内容，首先介绍了网站登录的原理，并讲解如何使用FormRequest提交登录表单模拟登录，然后讲解识别验证码的3种方法，最后介绍如何使用浏览器Cookie直接登录，并实现了一个下载中间件BrowserCookiesMiddleware。

第11章 爬取动态页面

在之前章节中，我们爬取的都是静态页面中的信息，静态页面的内容始终不变，爬取相对容易，但在现实中，目前绝大多数网站的页面都是动态页面，动态页面中的部分内容是浏览器运行页面中的JavaScript脚本动态生成的，爬取相对困难，这一章来学习如何爬取动态页面。

先来看一个简单的动态页面的例子，在浏览器中打开http://quotes.toscrape.com/js，显示如图11-1所示。

页面中有10条名人名言，每一条都包含在一个<div class="quote">元素中，如图11-2所示。现在，我们在scrapy shell环境下尝试爬取页面中的名人名言：

```
$ scrapy shell http://quotes.toscrape.com/js/
...
>>> response.css('div.quote')
[]
```

从结果看出，爬取失败了，在页面中没有找到任何包含名人名言的<div class="quote">元素。这些<div class="quote">就是动态内容，从服务器下载的页面中并不包含它们（所以我们爬取失败），浏览器执行了页面中的一段JavaScript代码后，它们才被生成出来，如图11-3所示。

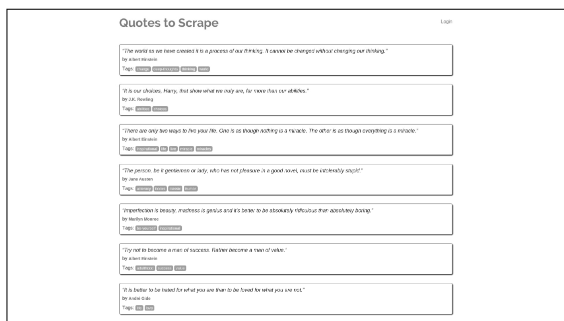


图11-1

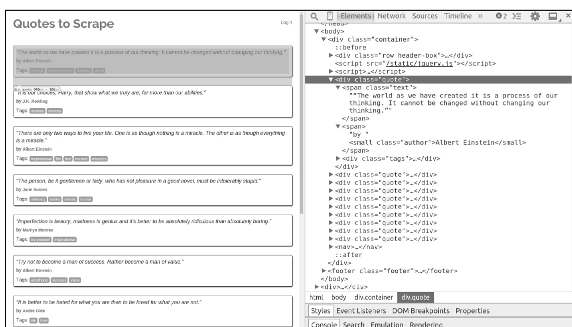


图11-2

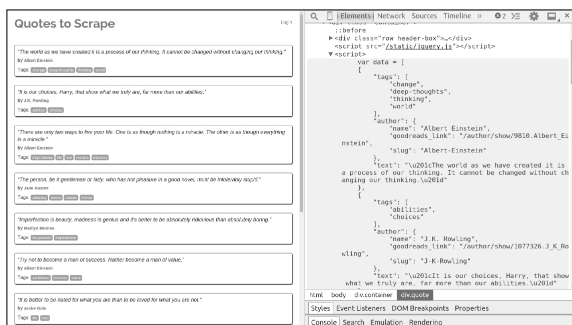


图11-3

图中的JavaScript代码如下：

```
var data = [
```

```
{
    "tags": [
        "change",
        "deep-thoughts",
        "thinking",
        "world"
    ],
    "author": {
        "name": "Albert Einstein",
        "goodreads_link": "/author/show/9810.Albert
        "slug": "Albert-Einstein"
    },
    "text": "\u201cThe world as we have created it
               It cannot be changed without changin
},
{
    "tags": [
        "abilities",
        "choices"
    ],
    "author": {
        "name": "J.K. Rowling",
        "goodreads_link": "/author/show/1077326.J_K
        "slug": "J-K-Rowling"
    },
    "text": "\u201cIt is our choices, Harry, that s
               far more than our abilities.\u201d"
},
... 省略部分内容 ...
{
    "tags": [
        "humor",
        "obvious",
        "simile"
    ],
    "author": {
        "name": "Steve Martin",
        "goodreads_link": "/author/show/7103.Steve_
        "slug": "Steve-Martin"
    },
```

```
        "text": "\u201cA day without sunshine is like,
    }
    ];
    for (var i in data) {
        var d = data[i];
        var tags = $.map(d['tags'], function(t) {
            return "<a class='tag'>" + t + "</a>";
        }).join(" ");
        document.write("<div class='quote'><span class=
            d['text'] + "</span><span>by <small class='
            d['author']['name'] + "</small></span><div
            tags + "</div></div>");
    }
```

阅读代码可以了解页面动态生成的细节，所有名人名言信息被保存在数组data中，最后的for循环迭代data中的每项信息，使用document.write生成每条名人名言对应的<div class="quote">元素。

上面是动态网页中最简单的一个例子，数据被硬编码于JavaScript代码中，实际中更常见的是JavaScript通过HTTP请求跟网站动态交互获取数据（AJAX），然后使用数据更新HTML页面。爬取此类动态网页需要先执行页面中的JavaScript代码渲染页面，再进行爬取。下面我们介绍如何使用JavaScript渲染引擎渲染页面。

11.1　Splash渲染引擎

Splash是Scrapy官方推荐的JavaScript渲染引擎，它是使用Webkit开发的轻量级无界面浏览器，提供基于HTTP接口的JavaScript渲染服务，支持以下功能：

- 为用户返回经过渲染的HTML页面或页面截图。
- 并发渲染多个页面。
- 关闭图片加载，加速渲染。
- 在页面中执行用户自定义的JavaScript代码。

- 执行用户自定义的渲染脚本（lua），功能类似于PhantomJS。

首先安装Splash，在linux下使用docker安装十分方便：

```
$ sudo apt-get install docker
$ sudo docker pull scrapinghub/splash
```

安装完成后，在本机的8050和8051端口开启Splash服务：

```
$ sudo docker run -p 8050:8050 -p 8051:8051 scrapi
[-] Log opened.
[-] Splash version: 2.1
[-] Qt 5.5.1, PyQt 5.5.1, WebKit 538.1, sip 4.17,
[-] Python 3.5 (default, Oct 14 2015, 20:28:29) [G
[-] Open files limit: 524288
[-] Open files limit increased from 524288 to 1048
[-] Xvfb is started: ['Xvfb', ':1', '-screen', '0'
[-] proxy profiles support is enabled, proxy profi
[-] verbosity=1
[-] slots=50
[-] argument_cache_max_entries=500
[-] Web UI: enabled, Lua: enabled (sandbox: enable
[-] Site starting on 8050
[-] Starting factory
```

Splash功能丰富，包含多个服务端点，由于篇幅有限，这里只介绍其中两个最常用的端点：

- render.html

  提供JavaScript页面渲染服务。

- execute

  执行用户自定义的渲染脚本（lua），利用该端点可在页面中执行JavaScript代码。

Splash文档地址：http://splash.readthedocs.io/en/latest/api.html。

11.1.1　render.html端点

JavaScript页面渲染服务是Splash中最基础的服务，请看表11-1中列出的文档。

表11-1

服务端点	render.html
请求地址	http://localhost:8050/render.html
请求方式	GET / POST
返回类型	html

render.html端点支持的参数如表11-2所示。

表11-2

参数	是否必选	类型	描述
url	必选	string	需要渲染页面的 url
timeout	可选	float	渲染页面超时时间
proxy	可选	string	代理服务器地址
wait	可选	float	等待页面渲染的时间
images	可选	integer	是否下载图片，默认为 1
js_source	可选	string	用户自定义的 JavaScript 代码，在页面渲染前执行

这里仅列出部分常用参数，详细内容参见官方文档。

下面是使用requests库调用render.html端点服务对页面http://quotes.toscrape.com/js/进行渲染的示例代码。

```
>>> import requests
>>> from scrapy.selector import Selector
>>> splash_url = 'http://localhost:8050/render.htm
    >>> args = {'url': 'http://quotes.toscrape.co
    >>> response = requests.get(splash_url, param
    >>> sel = Selector(response)
    >>> sel.css('div.quote span.text::text').extr
    ['"The world as we have created it is a proce
```

```
our thinking.”',
        '“It is our choices, Harry, that show what w
        '“There are only two ways to live your life.
though everything is a miracle.”',
        '“The person, be it gentleman or lady, who h
stupid.”',
        "“Imperfection is beauty, madness is genius
boring.”",
        '“Try not to become a man of success. Rather
        '“It is better to be hated for what you are
        "“I have not failed. I've just found 10,000
        "“A woman is like a tea bag; you never know
        '“A day without sunshine is like, you know,
```

在上述代码中，依据文档中的描述设置参数url、timeout、images，然后发送HTTP请求到服务接口地址。从运行结果看出，页面渲染成功，我们爬取到了页面中的10条名人名言。

11.1.2　execute端点

在爬取某些页面时，我们想在页面中执行一些用户自定义的JavaScript代码，例如，用JavaScript模拟点击页面中的按钮，或调用页面中的JavaScript函数与服务器交互，利用Splash的execute端点提供的服务可以实现这样的功能。请看表11-3中的文档。

表11-3

服务端点	execute
请求地址	http://localhost:8050/execute
请求方式	POST
返回类型	自定义

execute端点支持的参数如表11-4所示。

表11-4

参数	必选/可选	类型	描述
lua_source	必选	string	用户自定义的 lua 脚本
timeout	可选	float	渲染页面超时时间
proxy	可选	string	代理服务器地址

我们可以将execute端点的服务看作一个可用lua语言编程的浏览器，功能类似于PhantomJS。使用时需传递一个用户自定义的lua脚本给Splash，该lua脚本中包含用户想要模拟的浏览器行为，例如：

- 打开某url地址的页面
- 等待页面加载及渲染
- 执行JavaScript代码
- 获取HTTP响应头部
- 获取Cookie

下面是使用requests库调用execute端点服务的示例代码。

```
>>> import requests
>>> import json


>>> lua_script = '''
... function main(splash)
...   splash:go("http://example.com")
...   splash:wait(0.5)
...   local title = splash:evaljs("document.title"
...   return {title=title}
... end
... '''
>>> splash_url = 'http://localhost:8050/execute'
>>> headers = {'content-type': 'application/json'}
>>> data = json.dumps({'lua_source': lua_script})
>>> response = requests.post(splash_url, headers=h
>>> response.content
b'{"title": "Example Domain"}'
>>> response.json()
{'title': 'Example Domain'}
```

用户自定义的lua脚本中必须包含一个main函数作为程序入口，main函数被调用时会传入一个splash对象（lua中的对象），用户可以

调用该对象上的方法操纵Splash。例如，在上面的例子中，先调用go方法打开某页面，再调用wait方法等待页面渲染，然后调用evaljs方法执行一个JavaScript表达式，并将结果转化为相应的lua对象，最终Splash根据main函数的返回值构造HTTP响应返回给用户，main函数的返回值可以是字符串，也可以是lua中的表（类似Python字典），表会被编码成json串。

接下来，看一下splash对象常用的属性和方法。

- splash.args属性

  用户传入参数的表，通过该属性可以访问用户传入的参数，如splash.args.url、splash.args.wait。

- splash.js_enabled属性

  用于开启/禁止JavaScript渲染，默认为true。

- splash.images_enabled属性

  用于开启/禁止图片加载，默认为true。

- splash:go方法

  splash:go{url, baseurl=nil, headers=nil, http_method="GET", body=nil, formdata=nil}

  类似于在浏览器中打开某url地址的页面，页面所需资源会被加载，并进行JavaScript渲染，可以通过参数指定HTTP请求头部、请求方法、表单数据等。

- splash:wait方法

  splash:wait{time, cancel_on_redirect=false, cancel_on_error=true}等待页面渲染，time参数为等待的秒数。

- splash:evaljs方法

  splash:evaljs(snippet)

  在当前页面下，执行一段JavaScript代码，并返回最后一句

表达式的值。

- splash:runjs方法

  splash:runjs(snippet)

  在当前页面下，执行一段JavaScript代码，与evaljs方法相比，该函数只执行JavaScript代码，不返回值。

- splash:url方法

  splash:url()

  获取当前页面的url。

- splash:html方法

  splash:html()

  获取当前页面的HTML文本。

- splash:get_cookies方法

  splash:get_cookies()

  获取全部Cookie信息.

11.2　在Scrapy中使用Splash

掌握了Splash渲染引擎的基本使用后，我们继续学习如何在Scrapy中调用Splash服务，Python库的scrapy-splash是非常好的选择。

使用pip安装scrapy-splash：

```
$ pip install scrapy-splash
```

在项目环境中讲解scrapy-splash的使用，创建一个Scrapy项目，取名为splash_examples：

```
```

```
$ scrapy startproject splash_examples
```

首先在项目配置文件settings.py中对scrapy-splash进行配置，添加内容如下：

```
# Splash服务器地址
SPLASH_URL = 'http://localhost:8050'


# 开启Splash的两个下载中间件并调整HttpCompressionMiddl
DOWNLOADER_MIDDLEWARES = {
    'scrapy_splash.SplashCookiesMiddleware': 723,
    'scrapy_splash.SplashMiddleware': 725,
    'scrapy.downloadermiddlewares.httpcompression.H
}


# 设置去重过滤器
DUPEFILTER_CLASS = 'scrapy_splash.SplashAwareDupeF


# 用来支持cache_args（可选）
SPIDER_MIDDLEWARES = {
    'scrapy_splash.SplashDeduplicateArgsMiddleware'
}
```

编写Spider代码过程中，使用scrapy_splash调用Splash服务非常简单，scrapy_splash中定义了一个SplashRequest类，用户只需使用scrapy_splash.SplashRequest（替代scrapy.Request）提交请求即可。下面是SplashRequest构造器方法中的一些常用参数。

- url

  与scrapy.Request中的url相同，也就是待爬取页面的url（注意，不是Splash服务器地址）。

- headers

  与scrapy.Request中的headers相同。

- cookies

  与scrapy.Request中的cookies相同。

- args

  传递给Splash的参数（除url以外），如wait、timeout、images、js_source等。

- cache_args

  如果args中的某些参数每次调用都重复传递并且数据量较大（例如一段JavaScript代码），此时可以把该参数名填入cache_args列表中，让Splash服务器缓存该参数，如SplashRequest(url, args = {'js_source': js, 'wait': 0.5}, cache_args = ['js_source'])。

- endpoint

  Splash服务端点，默认为'render.html'，即JavaScript页面渲染服务，该参数可以设置为'render.json'、'render.har'、'render.png'、'render.jpeg'、'execute
  有些服务端点的功能我们没有讲解，详细内容可以查阅文档。

- splash_url

  Splash服务器地址，默认为None，即使用配置文件中SPLASH_URL的地址。

现在，大家已经对如何在Scrapy中使用Splash渲染引擎爬取动态页面有了一定了解，接下来我们在已经配置了Splash使用环境的splash_examples项目中完成两个实战项目。

11.3 项目实战：爬取toscrape中的名人名言

11.3.1 项目需求

爬取网站http://quotes.toscrape.com/js中的名人名言信息。

11.3.2 页面分析

该网站的页面已在本章开头部分分析过，大家可以回头看相关内容。

11.3.3 编码实现

首先，在splash_examples项目目录下使用scrapy genspider命令创建Spider：

```
scrapy genspider quotes quotes.toscrape.com
```

在这个案例中，我们只需使用Splash的render.html端点渲染页面，再进行爬取即可实现QuotesSpider，代码如下：

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy_splash import SplashRequest


class QuotesSpider(scrapy.Spider):
    name = "quotes"
    allowed_domains = ["quotes.toscrape.com"]
    start_urls = ['http://quotes.toscrape.com/js/']


    def start_requests(self):
        for url in self.start_urls:
            yield SplashRequest(url, args={'images'
```

```
    def parse(self, response):
        for sel in response.css('div.quote'):
            quote = sel.css('span.text::text').extr
            author = sel.css('small.author::text').
            yield {'quote': quote, 'author': author
        href = response.css('li.next > a::attr(href
        if href:
            url = response.urljoin(href)
            yield SplashRequest(url, args={'images'
```

上述代码中，使用SplashRequest提交请求，在SplashRequest的构造器中无须传递endpoint参数，因为该参数默认值便是'render.html'。使用args参数禁止Splash加载图片，并设置渲染超时时间。

运行爬虫，观察结果：

```
$ scrapy crawl quotes -o quotes.csv
...
     $ cat -n quotes.csv
          1   quote,author
          2   "The world as we have created it is a
changing our thinking.",Albert Einstein
          3   ""It is our choices, Harry, that show
Rowling
          4   "There are only two ways to live your
as though everything is a miracle.",Albert Einstei
          5   ""The person, be it gentleman or lady
intolerably stupid."",Jane Austen
          6   ""Imperfection is beauty, madness is
absolutely boring."",Marilyn Monroe
          7   "Try not to become a man of success.
          8   "It is better to be hated for what yo
          9   ""I have not failed. I've just found
          10  "A woman is like a tea bag; you never
Roosevelt
     ...
```

```
          91  ""I believe in Christianity as I beli
because by it I see everything else."",C.S. Lewis
          92  ""The truth."" Dumbledore sighed. ""I
be treated with great caution."",J.K. Rowling
          93  ""I'm the one that's got to die when
want to."",Jimi Hendrix
          94  "To die will be an awfully big advent
          95  "It takes courage to grow up and beco
          96  "But better to get hurt by the truth
          97  "You never really understand a person
Until you climb inside of his skin and walk around
          98  ""You have to write the book that wan
grown-ups, then you write it for children."",Madel
          99  "Never tell the truth to people who a
         100  ""A person's a person, no matter how
         101  ""... a mind needs books as a sword n
Martin
```

运行结果显示，我们成功爬取了10个页面中的100条名人名言。

11.4 项目实战：爬取京东商城中的书籍信息

11.4.1 项目需求

爬取京东商城中所有Python书籍的名字和价格信息。

11.4.2 页面分析

图11-4所示为在京东网站（http://www.jd.com）的书籍分类下搜索Python关键字得到的页面。



图11-4

结果有很多页，在每一个书籍列表页面中可以数出有60本书，但我们在scrapy shell中爬取该页面时遇到了问题，仅在页面中找到了30本书，少了30本，代码如下：

```
$ scrapy shell
...
>>> url = 'https://search.jd.com/Search?keyword=py
>>> fetch(url)
...
>>> len(response.css('ul.gl-warp > li'))
30
```

原来页面中的60本书不是同时加载的，开始只有30本书，当我们使用鼠标滚轮滚动到页面下方某位置时，后30本书才由JavaScript脚本加载，通过实验可以验证这个说法，实验流程如下：

（1）页面刚加载时，在Chrome开发者工具的console中用jQuery代码查看当前有多少本书，此时为30。

（2）之后滚动鼠标滚轮到某一位置时，可以看到JavaScript发送HTTP请求和服务器交互（XHR）。

（3）然后用jQuery代码查看当前有多少本书，已经变成了60，如图11-5所示。



图11-5

既然如此，爬取这个页面时，可以先执行一段JavaScript代码，将滚动条拖到页面下方某位置，触发加载后30本书的事件，在开发者工具的console中进行实验，用document.getElementsByXXX方法随意选中页面下方的某元素，比如“下一页”按钮所在的<div>元素，然后在该元素对象上调用scrollIntoView(true)完成拖曳动作，此时查看书籍数量，变成了60，这个解决方案是可行的。图11-6所示为实验过程。

爬取一个页面的问题解决了，再来研究如何从页面中找到下一页的url地址。



图11-6

如图11-7所示，下一页链接的href属性并不是一个url，而在其onclick属性中包含了一条JavaScript代码，单击“下一页”按钮时会调用函数SEARCH.page(n,　true)。虽然可以用Splash执行函数来跳转到下一页，但还是很麻烦，经观察发现，每个页面url的差异仅在于page

参数不同，第一页page＝1，第2页page＝3，第3页page＝5……以2递增，并且页面中还包含商品总数信息。因此，我们可以推算出所有页面的url。

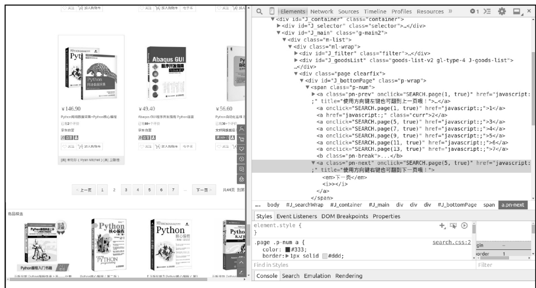


图11-7

11.4.3　编码实现

首先，在splash_examples项目目录下使用scrapy　genspider命令创建Spider类：

```
scrapy genspider jd_book search.jd.com
```

经上述分析，在爬取每一个书籍列表页面时都需要执行一段JavaScript代码，以让全部书籍加载，因此选用execute端点完成该任务，实现JDBookSpider代码如下：

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy import Request
from scrapy_splash import SplashRequest


lua_script = '''
function main(splash)
   splash:go(splash.args.url)
```

```
    splash:wait(2)
    splash:runjs("document.getElementsByClassName('
    splash:wait(2)
    return splash:html()
end
'''


class JDBookSpider(scrapy.Spider):
    name = "jd_book"
    allowed_domains = ["search.jd.com"]
    base_url = 'https://search.jd.com/Search?keywor


    def start_requests(self):
        # 请求第一页，无须js渲染
        yield Request(self.base_url, callback=self.


    def parse_urls(self, response):
        # 获取商品总数，计算出总页数
        total = int(response.css('span#J_resCount::
        pageNum = total // 60 + (1 if total % 60 el
        # 构造每页的url，向Splash的execute 端点发送请求
        foriin range(pageNum):
            url = '%s&page=%s' % (self.base_url, 2
            yield SplashRequest(url,
                endpoint='execute',
                args={'lua_source': lua_script},
                cache_args=['lua_source'])


def parse(self, response):
    # 获取一个页面中每本书的名字和价格
    for sel in response.css('ul.gl-warp.clearfix >
        yield {
            'name': sel.css('div.p-name').xpath('str
            'price': sel.css('div.p-price i::text').
        }
```

解释上述代码如下：

- start_requests方法

  start_requests提交对第一个页面的请求，这个页面不需要渲染，因为我们只想从中获取页面总数，使用scrapy.Request提交请求，并指定parse_urls作为解析函数。

- parse_urls方法

  从第一个页面中提取商品总数，用其计算页面总数，之后按照前面分析出的页面url规律构造每一个页面的url。这些页面都是需要渲染的，使用SplashRequest提交请求，除了渲染页面以外，还需要执行一段JavaScript代码（为了加载后30本书），因此使用Splash的execute端点将endpoint参数置为'execute'。通过args参数的lua_source字段传递我们要执行的lua脚本，由于爬取每个页面时都要执行该脚本，因此可以使用cache_args参数将该脚本缓存到Splash服务器。

- parse方法

  一个页面中提取60本书的名字和价格信息，相关内容大家早已熟悉，不再赘述。

- lua_script字符串

  自定义的lua脚本，其中的逻辑很简单：

  打开页面→等待渲染→执行js触发数据加载（后30本书）→等待渲染→返回html。

另外，京东服务器程序会对请求头部中的User-Agent字段进行检测，因此需要在配置文件settings.py中设置USER_AGENT，伪装成常规浏览器：

```
# 伪装成常规浏览器
USER_AGENT = 'Mozilla/5.0 (X11; Linux x86_64) Appl
```

编码和配置的工作已经完成了，运行爬虫并观察结果：

```
$ scrapy crawl jd_book -o books.csv
...
$ cat -n books.scv
     1    name,price
     2    教孩子学编程 Python语言版,59.00
     3    Python机器学习 预测分析核心算法,46.90
     4    Python机器学习 预测分析核心算法 9787115433732
     5    Python数据分析实战 9787115432209,39.50
     6    数据科学实战手册 R+Python 9787115426758,39.50
     7    Python编程 从入门到实践,89.00
     8    Python可以这样学,69.00
     9    Python游戏编程快速上手,47.20
    10    Python性能分析与优化,45.00
   ...
  2932    Python语言在Abaqus 中的应用（附CD-ROM光盘1 张）
  2933    Python绝技 运用Python成为*级黑客 运用Python成为
  2934    趣学Python编程,42.70
  2935    零基础学Python（图文版）,65.50
  2936    Python程序设计入门到实战,65.50
  2937    计算机科学丛书：Python语言程序设计,74.60
  2938    面向ArcGIS的Python脚本编程,41.00
  2939    Python算法教程 书籍,40.60
  2940    Python基础教程+利用Python进行数据分析 Python学
  2941    包邮 量化投资 以Python 为工具 Python语言处理数据
```

结果显示，我们成功爬取到了2940本书籍的信息。

11.5 本章小结

本章学习了爬取动态页面的相关知识，首先带大家了解了动态页面的实现原理，然后介绍了页面渲染引擎Splash，并详细讲解了其中render.html和execute两个服务端点的使用，最后通过两个案例展示了在Scrapy爬虫中如何利用Splash服务爬取动态页面的内容。

第12章
存入数据库

在之前的章节中，曾讨论过将爬取到的数据导出到文件的相关话题，但在某些时候，我们希望将爬取到的数据存储到数据库中，这一章来学习使用Item　Pipeline实现Scrapy爬虫和几种常用数据库的接口。

以第8章toscrape_book项目作为环境展开本章内容的讲解。在toscrape_book项目中，我们爬取了网站http://books.toscrape.com中的书籍信息，其中每一本书的信息包括：

- 书名
- 价格
- 评价等级
- 产品编码
- 库存量
- 评价数量

下面我们来学习如何在爬取数据的过程中将书籍信息存储到各种数据库，这些数据库主要有：SQLite、MySQL、MongoDB、Redis。

12.1　SQLite

SQLite是一个文件型轻量级数据库，它的处理速度很快，在数据量不是很大的情况下，使用SQLite足够了。

首先，创建一个供Scrapy使用的SQLite数据库，取名为scrapy.db：

```
$ sqlite3 scrapy.db
```

```
...
sqlite>
```

接下来，在客户端中创建数据表（Table）：

```
CREATE TABLE books (
   upc                  CHAR(16) NOT NULL PRIMARY KEY
   name                 VARCHAR(256) NOT NULL,
   price                VARCHAR(16) NOT NULL,
   review_rating        INT,
   review_num           INT,
   stock                INT
);
```

在Python中访问SQLite数据库可使用Python标准库中的sqlite3模块。下面是使用sqlite3模块将数据写入SQLite数据库的简单示例：

```
import sqlite3

#连接数据库，得到Connection 对象
conn = sqlite3.connect('example.db')

#创建Curosr 对象，用来执行SQL语句
cur = conn.cursor()

#创建数据表
cur.execute('CREATE TABLE person (name VARCHAR(32)

#插入一条数据
cur.execute('INSERT INTO person VALUES (?,?,?)', (
#保存变更, commit 后数据才被实际写入数据库
conn.commit()
```

```
#关闭连接
conn.close()
```

了解了在Python中如何操作SQLite数据库后，接下来编写一个能将爬取到的数据写入SQLite数据库的Item Pipeline。在pipelines.py中实现SQLitePipeline的代码如下：

```
import sqlite3

class SQLitePipeline(object):

   def open_spider(self, spider):
      db_name = spider.settings.get('SQLITE_DB_NAM

      self.db_conn = sqlite3.connect(db_name)
      self.db_cur = self.db_conn.cursor()

   def close_spider(self, spider):
      self.db_conn.commit()
      self.db_conn.close()

   def process_item(self, item, spider):
      self.insert_db(item)

      return item

   def insert_db(self, item):
      values = (
         item['upc'],
```

```
            item['name'],
            item['price'],
            item['review_rating'],
            item['review_num'],
            item['stock'],
        )


        sql = 'INSERT INTO books VALUES (?,?,?,?,?,?
self.db_cur.execute(sql, values)


# 每插入一条就commit一次会影响效率
# self.db_conn.commit()
```

解释上述代码如下：

- open_spider方法在开始爬取数据之前被调用，在该方法中通过spider.settings对象读取用户在配置文件中指定的数据库，然后建立与数据库的连接，将得到的Connection对象和Cursor对象分别赋值给self.db_conn和self.db_cur，以便之后使用。
- process_item方法处理爬取到的每一项数据，在该方法中调用insert_db方法，执行插入数据操作的SQL语句。但需要注意的是，在insert_db中并没有调用连接对象的commit方法，也就意味着此时数据并没有实际写入数据库。如果每插入一条数据都调用一次commit方法，会严重降低程序执行效率，并且我们对数据插入数据库的实时性并没有什么要求，因此可以在爬取完全部数据后再调用commit方法。
- close_spider方法在爬取完全部数据后被调用，在该方法中，调用连接对象的commit方法将之前所有的插入数据操作一次性提交给数据库，然后关闭连接对象。

在配置文件settings.py中指定我们所要使用的SQLite数据库，并启用SQLitePipeline：

```
SQLITE_DB_NAME = 'scrapy.db'

ITEM_PIPELINES = {
   'toscrape_book.pipelines.SQLitePipeline': 400,
}
```

运行爬虫，并查看数据库：

```
$ scrapy crawl books
...
$ sqlite3 scrapy.db
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select count(*) from books;
1000
sqlite> select * from books;
a22124811bfa8350|It's Only the Himalayas|£45.17|2
feb7cc7701ecf901|Olio|£23.88|1|0|19
a34ba96d4081e6a4|Rip it Up and Start Again|£35.02
a18a4f574854aced|Libertarianism for Beginners|£51
ce6396b0f23f6ecc|Set Me Free|£17.46|5|0|19
fa9610a50a1bf149|Masks and Shadows|£56.40|2|0|16
3c346ab1e76ae1f6|Obsidian (Lux #1)|£14.86|2|0|16
09b6cc87e62c2c58|Danganronpa Volume 1|£51.99|4|0|
...
```

结果表明，我们成功地将1000条数据存储到了SQLite数据库。

12.2　MySQL

MySQL是一个应用极其广泛的关系型数据库，它是开源免费的，可以支持大型数据库，在个人用户和中小企业中成为技术首选。

使用客户端登录MySQL，创建一个供Scrapy使用的数据库，取名

为scrapy_db：

```
$ mysql -hlocalhost -uliushuo -p12345678
...
mysql> CREATE DATABASE scrapy_db CHARACTER SET 'ut
Query OK, 1 row affected (0.00 sec)
mysql> USE scrapy_db;
Database changed
```

接下来，创建存储书籍数据的表：

```
mysql> CREATE TABLE books (
    ->       upc                  CHAR(16) NOT NULL PRI
    ->       name                 VARCHAR(256) NOT NULL
    ->       price               VARCHAR(16) NOT NULL,
    ->       review_rating   INT,
    ->       review_num         INT,
    ->       stock               INT
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.08 sec)
```

在Python 2中访问MySQL数据库可以使用第三方库MySQL-Python（即MySQLdb），但是MySQLdb不支持Python 3。在Python 3中，可以使用另一个第三方库mysqlclient作为替代，它是基于MySQL-Python开发的，提供了几乎完全相同的接口。因此，在两个Python版本下，可以使用相同的代码访问MySQL。

Python 2使用pip安装MySQL-python：

```
sudo pip install MySQL-python
```

Python 3使用pip安装mysqlclient：

```

```

```
sudo pip install mysqlclient
```

下面是使用MySQLdb将数据写入MySQL数据库的简单示例，与sqlite3的使用几乎完全相同：

```
import MySQLdb

#连接数据库，得到Connection 对象
conn = MySQLdb.connect(host='localhost', db='scrap
            user='liushuo', passwd='12345678', cha

#创建Curosr 对象，用来执行SQL语句
cur = conn.cursor()

#创建数据表
cur.execute('CREATE TABLE person (name VARCHAR(32)
            ENGINE=InnoDB DEFAULT CHARSET=utf8')

#插入一条数据
cur.execute('INSERT INTO person VALUES (%s,%s,%s)'

#保存变更，commit 后数据才被实际写入数据库
conn.commit()

#关闭连接
conn.close()
```

仿照SQLitePipeline实现MySQLPipeline，代码如下：

```
import MySQLdb
```

```
class MySQLPipeline:
   def open_spider(self, spider):
       db = spider.settings.get('MYSQL_DB_NAME', '
       host = spider.settings.get('MYSQL_HOST', 'l
   port = spider.settings.get('MYSQL_PORT', 3306)
   user = spider.settings.get('MYSQL_USER', 'root'
   passwd = spider.settings.get('MYSQL_PASSWORD',


   self.db_conn = MySQLdb.connect(host=host, port=
                             user=user, passwd=passwd,
   self.db_cur = self.db_conn.cursor()


def close_spider(self, spider):
   self.db_conn.commit()
   self.db_conn.close()


def process_item(self, item, spider):
   self.insert_db(item)


   return item


def insert_db(self, item):
   values = (
       item['upc'],
       item['name'],
       item['price'],
       item['review_rating'],
       item['review_num'],
       item['stock'],
   )


   sql = 'INSERT INTO books VALUES (%s,%s,%s,%s,%s
```

```
        self.db_cur.execute(sql, values)
```

上述代码结构与SQLitePipeline完全相同，不再赘述。

在配置文件settings.py中指定我们所要使用的MySQL数据库，并启用MySQLPipeline：

```
MYSQL_DB_NAME = 'scrapy_db'
MYSQL_HOST = 'localhost'
MYSQL_USER = 'liushuo'
MYSQL_PASSWORD = '12345678'


ITEM_PIPELINES = {
  'toscrape_book.pipelines.MySQLPipeline': 401,
}
```

运行爬虫，并查看数据库：

```
$ scrapy crawl books
...
$ mysql -hlocalhost -uliushuo -p12345678 scrapy_db
...
mysql> select count(*) from books;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)
mysql> select name from books;
+--------------------------------------------------------------------+
| name                                                               |
+--------------------------------------------------------------------+
| Deep Under (Walker Security #1)                                    |
| Black Dust                                                         |
| The Songs of the Gods                                              |
| The Passion of Dolssa                                              |
| The Rosie Project (Don Tillman #1)                                 |
| Give It Back                                                       |
| A Brush of Wings (Angels Walking #3)                               |
| A Court of Thorns and Roses (A Court of Thorns and Roses #1)       |
| You (You #1)                                                       |
| Travels with Charley: In Search of America                         |
| Old School (Diary of a Wimpy Kid #10)                              |
| Starving Hearts (Triangular Trade Trilogy, #1)                     |
| The Origin of Species                                              |
| Is Everyone Hanging Out Without Me? (And Other Concerns)           |
| A Game of Thrones (A Song of Ice and Fire #1)                      |
| Living Forward: A Proven Plan to Stop Drifting and Get the Life    |
```

```
...省略中间输出...

| The Perks of Being a Wallflower                                    |
| The Vacationers                                                    |
| My Perfect Mistake (Over the Top #1)                               |
| I Had a Nice Time And Other Lies...: How to find love              |
| In Cold Blood                                                      |
| Still Life with Bread Crumbs                                       |
| The Expatriates                                                    |
| Soldier (Talon #3)                                                 |
| Olio                                                               |
| The Dream Thieves (The Raven Cycle #2)                             |
| A Murder Over a Girl: Justice, Gender, Junior High                 |
| The Demonists (Demonist #1)                                        |
| The Husband's Secret                                               |
| Amatus                                                             |
| Art Ops Vol. 1                                                     |
| The Haters                                                         |
+--------------------------------------------------------------------+
```

结果表明，我们成功地将1000条数据存储到了MySQL数据库。

上述代码中，同样是先执行完全部的插入语句（INSERT INTO），最后一次性调用commit方法提交给数据库。或许在某些情况下，我们的确需要每执行一条插入语句，就立即调用commit方法更新数据库，如爬取过程很长，中途可能被迫中断，这样程序就不能执行到最后的commit。如果在上述代码的insert_db方法中直接添加self.db_conn.commit()，又会使程序执行慢得让人无法忍受。为解决以上难题，下面讲解另一种实现方法。

Scrapy框架自身是使用另一个Python框架Twisted编写的程序，Twisted是一个事件驱动型的异步网络框架，鼓励用户编写异步代码，Twisted中提供了以异步方式多线程访问数据库的模块adbapi，使用该模块可以显著提高程序访问数据库的效率。下面是使用adbapi中的连接池访问MySQL数据库的简单示例：

```
from twisted.internet import reactor, defer
from twisted.enterprise import adbapi
import threading


dbpool = adbapi.ConnectionPool('MySQLdb', host='lo
                               user='liushuo', password=
def insert_db(tx, item):
   print('In Thread:', threading.get_ident())
   sql = 'INSERT INTO person VALUES (%s, %s, %s)'
   tx.execute(sql, item)


for i in range(1000):
   item = ('person%s' % i, 25, 'M')
   dbpool.runInteraction(insert_db, item)


reactor.run()
```

上述代码解释如下：

- adbapi.ConnectionPool方法可以创建一个数据库连接池对象，其中包含多个连接对象，每个连接对象在独立的线程中工作。adbapi只是提供了异步访问数据库的编程框架，在其内部依然使用MySQLdb、sqlite3这样的库访问数据库。ConnectionPool方法的第一个参数就是用来指定使用哪个库访问数据库，其他参数在创建连接对象时使用。
- dbpool.runInteraction(insert_db, item)以异步方式调用instert_db函数，dbpool会选择连接池中的一个连接对象在独立线程中调用insert_db，其中参数item会被传给insert_db的第二个参数，传给insert_db的第一个参数是一个Transaction对象，其接口与Cursor对象类似，可以调用execute方法执行SQL语句，insert_db执行完后，连接对象会自动调用commit方法。

了解了adbapi的使用后，给出第二个版本的MySQLPipeline，代码如下：

```
from twisted.enterprise import adbapi


class MySQLAsyncPipeline:
   def open_spider(self, spider):
      db = spider.settings.get('MYSQL_DB_NAME', 's
      host = spider.settings.get('MYSQL_HOST', 'lo
      port = spider.settings.get('MYSQL_PORT', 330
      user = spider.settings.get('MYSQL_USER', 'ro
      passwd = spider.settings.get('MYSQL_PASSWORD


      self.dbpool = adbapi.ConnectionPool('MySQLdb
                                   user=user, pass
def close_spider(self, spider):
   self.dbpool.close()
```

```
    def process_item(self, item, spider):
        self.dbpool.runInteraction(self.insert_db, item

        return item

    def insert_db(self, tx, item):
        values = (
            item['upc'],
            item['name'],
            item['price'],
            item['review_rating'],
            item['review_num'],
            item['stock'],
        )

        sql = 'INSERT INTO books VALUES (%s,%s,%s,%s,%s
        tx.execute(sql, values)
```

通过前面的讲述，相信大家可以轻松理解上述代码，不再过多解释，该版本比之前的版本在执行效率上有显著提高。

12.3 MongoDB

MongoDB是一个面向文档的非关系型数据库（NoSQL），它功能强大、灵活、易于拓展，近年来在多个领域得到广泛应用。

在Python中可以使用第三方库pymongo访问MongoDB数据库，使用pip安装pymongo：

```
$ sudo pip install pymongo
```

下面是使用pymongo将数据写入MongoDB数据库的简单示例：

```
from pymongo import MongoClient
# 连接MongoDB，得到一个客户端对象
client = MongoClient('mongodb://localhost:27017')


# 获取名为scrapy_db的数据库的对象
db = client.scrapy_db


# 获取名为person的集合的对象
collection = db.person


doc = {
   'name': '刘硕',
   'age': 34,
   'sex': 'M',
}


# 将文档插入集合
collection.insert_one(doc)


# 关闭客户端
client.close()
```

仿照SQLitePipeline实现MongoDBPipeline，代码如下：

```
from pymongo import MongoClient
from scrapy import Item


class MongoDBPipeline:
   def open_spider(self, spider):
       db_uri = spider.settings.get('MONGODB_URI',
       db_name = spider.settings.get('MONGODB_DB_N
```

```
        self.db_client = MongoClient('mongodb://loc
        self.db = self.db_client[db_name]


    def close_spider(self, spider):
        self.db_client.close()


    def process_item(self, item, spider):
        self.insert_db(item)
    return item


  def insert_db(self, item):
    if isinstance(item, Item):
        item = dict(item)


    self.db.books.insert_one(item)
```

解释上述代码如下：

- open_spider方法在开始爬取数据之前被调用，在该方法中通过spider.settings对象读取用户在配置文件中指定的数据库，然后建立与数据库的连接，将得到的MongoClient对象和Database对象分别赋值给self.db_client和self.db，以便之后使用。
- process_item方法处理爬取到的每一项数据，在该方法中调用insert_db方法，执行数据库的插入操作。在insert_db方法中，先将一项数据转换成字典，然后调用insert_one方法将其插入集合books。
- close_spider方法在爬取完全部数据后被调用，在该方法中关闭与数据库的连接。

在配置文件settings.py中指定我们所要使用的MongoDB数据库，

并启用MongoDBPipeline：

```
MONGODB_URI = 'mongodb://localhost:27017'
MONGODB_DB_NAME = 'scrapy_db'


ITEM_PIPELINES = {
  'toscrape_book.pipelines.MongoDBPipeline': 403,
}
```

运行爬虫，并查看数据库：

```
$ scrapy crawl books
...
$ mongo scrapy_db
MongoDB shell version: 2.4.9
connecting to: scrapy_db
> db.books.count()
1000
> db.books.find()
     { "_id" : ObjectId("58fb48859dcd1928b736ee4f"
"22", "upc" : "a897fe39b1053632", "price" : "£51.
     { "_id" : ObjectId("58fb48859dcd1928b736ee50"
"19", "upc" : "feb7cc7701ecf901", "price" : "£23.
     { "_id" : ObjectId("58fb48859dcd1928b736ee51"
"19", "upc" : "a18a4f574854aced", "price" : "£51.
     { "_id" : ObjectId("58fb48859dcd1928b736ee52"
"19", "upc" : "e30f54cea9b38190", "price" : "£37.
Stories 1800-1849" }
     { "_id" : ObjectId("58fb48859dcd1928b736ee53"
"19", "upc" : "a34ba96d4081e6a4", "price" : "£35.
     { "_id" : ObjectId("58fb48859dcd1928b736ee54"
"19", "upc" : "a22124811bfa8350", "price" : "£45.
     { "_id" : ObjectId("58fb48859dcd1928b736ee55"
"19", "upc" : "3b1c02bac2a429e6", "price" : "£52.
Pilgrim #1)" }
     { "_id" : ObjectId("58fb48859dcd1928b736ee56"
```

```
"19", "upc" : "deda3e61b9514b83", "price" : "£57.
from the American Indie Underground, 1981-1991" }
    { "_id" : ObjectId("58fb48869dcd1928b736ee57"
"19", "upc" : "ce6396b0f23f6ecc", "price" : "£17.
    { "_id" : ObjectId("58fb48869dcd1928b736ee58"
"19", "upc" : "30a7f60cd76ca58c", "price" : "£20.
    { "_id" : ObjectId("58fb48869dcd1928b736ee59"
"19", "upc" : "0312262ecafa5a40", "price" : "£13.
#1)" }
    { "_id" : ObjectId("58fb48869dcd1928b736ee5a"
"19", "upc" : "1dfe412b8ac00530", "price" : "£52.
    { "_id" : ObjectId("58fb48869dcd1928b736ee5b"
"19", "upc" : "e10e1e165dc8be4a", "price" : "£22.
Their Epic Quest for Gold at the 1936 Berlin Olymp
    { "_id" : ObjectId("58fb48869dcd1928b736ee5c"
"19", "upc" : "f77dbf2323deb740", "price" : "£22.
    { "_id" : ObjectId("58fb48869dcd1928b736ee5d"
"19", "upc" : "2597b5a345f45e1b", "price" : "£33.
Dream Job" }
    { "_id" : ObjectId("58fb48869dcd1928b736ee5e"
"19", "upc" : "e72a5dfc7e9267b2", "price" : "£17.
the Life of the Infamous Feminist, Victoria Woodhu
    { "_id" : ObjectId("58fb48869dcd1928b736ee5f"
"20", "upc" : "4165285e1663650f", "price" : "£54.
    { "_id" : ObjectId("58fb48869dcd1928b736ee60"
"20", "upc" : "e00eb4fd7b871a48", "price" : "£47.
    { "_id" : ObjectId("58fb48869dcd1928b736ee61"
"20", "upc" : "90fa61229261140a", "price" : "£53.
    { "_id" : ObjectId("58fb48869dcd1928b736ee62"
"20", "upc" : "6957f44c3847a760", "price" : "£50.
    Type "it" for more
    >
```

结果表明，我们成功地将1000条数据存储到了MongoDB数据库。

12.4 Redis

Redis是一个使用ANSI C编写的高性能Key-Value数据库，使用内存作为主存储，内存中的数据也可以被持久化到硬盘。

在Python中可以使用第三方库redis-py访问Redis数据库，使用pip安装redis-py：

```
$ sudo pip install redis
```

下面是使用redis-py将数据写入Redis数据库的简单示例：

```
import redis


# 连接数据库
r = redis.StrictRedis(host='localhost', port=6379,


# 创建3 条数据
person1 = {
    'name': '刘硕',
    'age': 34,
    'sex': 'M',
}


person2= {
    'name': '李雷',
    'age': 32,
    'sex': 'M',
}


person3= {
    'name': '韩梅梅',
    'age': 31,
    'sex': 'F',
}
```

```
# 将3 条数据以Hash 类型（哈希）保存到Redis中
r.hmset('person:1', person1)
r.hmset('person:2', person2)
r.hmset('person:3', person3)

# 关闭连接
r.connection_pool.disconnect()
```

Redis是Key-Value数据库，一项数据在数据库中就是一个键值对，存储多项同类别的数据时（如Book），通常以item:id这样的形式作为每项数据的键，其中的“:”并没有什么特殊，也可以换成“-”或“/”等，只是大家习惯这样使用。

仿照SQLitePipeline实现RedisPipeline，代码如下：

```
import redis
from scrapy import Item


class RedisPipeline:
    def open_spider(self, spider):
        db_host = spider.settings.get('REDIS_HOST',
        db_port = spider.settings.get('REDIS_PORT',
        db_index = spider.settings.get('REDIS_DB_IND

        self.db_conn = redis.StrictRedis(host=db_hos
        self.item_i = 0

    def close_spider(self, spider):
        self.db_conn.connection_pool.disconnect()
def process_item(self, item, spider):
    self.insert_db(item)
```

```
        return item

    def insert_db(self, item):
        if isinstance(item, Item):
            item = dict(item)

        self.item_i += 1
        self.db_conn.hmset('book:%s' % self.item_i, ite
```

解释上述代码如下：

- open_spider方法在开始爬取数据之前被调用，在该方法中通过spider.settings对象读取用户在配置文件中指定的数据库，然后建立与数据库的连接，将得到的连接对象赋值给self.db_conn，以便之后使用，并初始化一个self.item_i作为每项数据的id。在插入一项数据时，使用self.item_i自加的结果构造数据在数据库中的键。
- process_item方法处理爬取到的每一项数据，在该方法中调用insert_db方法执行数据库的插入操作，在insert_db方法中先将一项数据转换成字典，然后调用hmset方法将数据以Hash类型存入Redis数据库。
- close_spider方法在爬取完全部数据后被调用，在该方法中关闭与数据库的连接。

在配置文件settings.py中指定我们所要使用的Redis数据库，并启用RedisPipeline：

```
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB_INDEX = 0
```

```
ITEM_PIPELINES = {
  'toscrape_book.pipelines.RedisPipeline': 404,
}
```

运行爬虫，并查看数据库：

```
$ scrapy crawl books
$ redis-cli
127.0.0.1:6379> KEYS book:*
  1) "book:470"
  2) "book:300"
  3) "book:801"
  4) "book:476"
  5) "book:914"


 ...省略中间输出...


 995) "book:703"
 996) "book:407"
 997) "book:995"
 998) "book:569"
 999) "book:298"
1000) "book:110"
127.0.0.1:6379> HGETALL book:1
 1) "price"
 2) "\xc2\xa351.33"
 3) "review_rating"
 4) "2"
 5) "review_num"
 6) "0"
 7) "name"
 8) "Libertarianism for Beginners"
 9) "stock"
10) "19"
11) "upc"
12) "a18a4f574854aced"
```

```
127.0.0.1:6379> HGETALL book:2
 1) "price"
 2) "\xc2\xa317.46"
 3) "review_rating"
 4) "5"
 5) "review_num"
 6) "0"
 7) "name"
 8) "Set Me Free"
 9) "stock"
10) "19"
11) "upc"
12) "ce6396b0f23f6ecc"
```

结果表明，我们成功地将1000条数据存储到了Redis数据库。

12.5　本章小结

本章学习了如何将爬取到的数据存储到数据库的相关内容，在Scrapy中可以实现Item　　Pipeline完成数据库存储的任务，我们以SQLite、MySQL、MongoDB和Redis几种常用数据库为例，先讲解在Python中将数据写入各种数据库的方法，然后实现相应的Item Pipeline。

第13章 使用HTTP代理

大家可能都有过给浏览器设置HTTP代理的经验，HTTP代理服务器可以比作客户端与Web服务器（网站）之间的一个信息中转站，客户端发送的HTTP请求和Web服务器返回的HTTP响应通过代理服务器转发给对方，如图13-1所示。

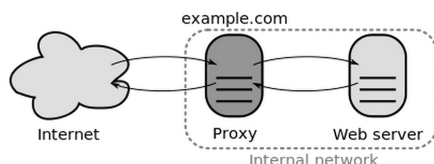


图13-1

爬虫程序在爬取某些网站时也需要使用代理，例如：

- 由于网络环境因素，直接爬取速度太慢，使用代理提高爬取速度。
- 某些网站对用户的访问速度进行限制，爬取过快会被封禁ip，使用代理防止被封禁。
- 由于地方法律或政治原因，某些网站无法直接访问，使用代理绕过访问限制。

这一章我们来学习Scrapy爬虫如何使用代理进行爬取。

13.1 HttpProxyMiddleware

Scrapy内部提供了一个下载中间件HttpProxyMiddleware，专门用于给Scrapy爬虫设置代理。

13.1.1 使用简介

HttpProxyMiddleware默认便是启用的，它会在系统环境变量中搜索当前系统代理（名字格式为xxx_proxy的环境变量），作为Scrapy爬虫使用的代理。

假设我们现在有两台在云上搭建好的代理服务器：

http://116.29.35.201:8118

http://197.10.171.143:8118

为本机的Scrapy爬虫分别设置发送HTTP和HTTPS请求时所使用的代理，只需在bash中添加环境变量：

```
$ export http_proxy="http://116.29.35.201:8118"
$ export https_proxy="http://197.10.171.143:8118"
```

配置完成后，Scrapy爬虫将会使用上面指定的代理下载页面，我们可以通过以下实验进行验证。

利用网站http://httpbin.org提供的服务可以窥视我们所发送的HTTP(S)请求，如请求源IP地址、请求头部、Cookie信息等。图13-2展示了该网站各种服务的API地址。

访问http(s)://httpbin.org/ip将返回一个包含请求源IP地址信息的json串，在scrapy shell中访问该url，查看请求源IP地址：

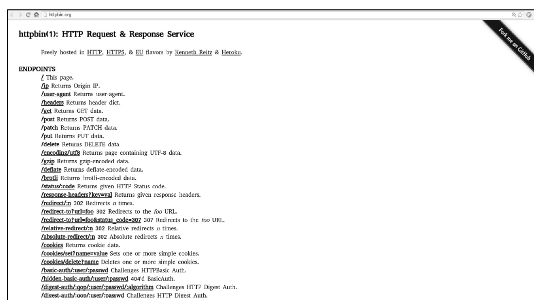


图13-2

```
$ scrapy shell
...
>>> import json
>>> fetch(scrapy.Request('http://httpbin.org/ip'))
[scrapy] DEBUG: Crawled (200)    (referer: None)
>>> json.loads(response.text)
{'origin': '116.29.35.201'}
>>> fetch(scrapy.Request('https://httpbin.org/ip')
[scrapy] DEBUG: Crawled (200)    (referer: None)
>>> json.loads(response.text)
{'origin': '197.10.171.143'}
```

在上述实验中，分别以HTTP和HTTPS发送请求，使用json模块对响应结果进行解析，读取请求源IP地址（origin字段），其值正是代理服务器的IP。由此证明，Scrapy爬虫使用了我们指定的代理。

上面我们使用的是无须身份验证的代理服务器，还有一些代理服务器需要用户提供账号、密码进行身份验证，验证成功后才提供代理服务，使用此类代理时，可按以下格式配置：

```
$ export http_proxy="http://liushuo:12345678@113.2
```

13.1.2　源码分析

虽然使用HttpProxyMiddleware很简单，但大家最好对其工作原理有所了解，以下是HttpProxyMiddleware的源码：

```
import base64
from six.moves.urllib.request import getproxies, p
from six.moves.urllib.parse import unquote
try:
   from urllib2 import _parse_proxy
except ImportError:
   from urllib.request import _parse_proxy
from six.moves.urllib.parse import urlunparse
```

```
from scrapy.utils.httpobj import urlparse_cached
from scrapy.exceptions import NotConfigured
from scrapy.utils.python import to_bytes


class HttpProxyMiddleware(object):


   def __init__(self, auth_encoding='latin-1'):
      self.auth_encoding = auth_encoding
      self.proxies = {}
      for type, url in getproxies().items():
         self.proxies[type] = self._get_proxy(url,


      if not self.proxies:
         raise NotConfigured


   @classmethod
   def from_crawler(cls, crawler):
      auth_encoding = crawler.settings.get('HTTPPR
      return cls(auth_encoding)
   def _get_proxy(self, url, orig_type):
      proxy_type, user, password, hostport = _pars
      proxy_url = urlunparse((proxy_type or orig_t
      if user:
         user_pass = to_bytes(
            '%s:%s' % (unquote(user), unquote(pass
            encoding=self.auth_encoding)
       creds = base64.b64encode(user_pass).strip()
   else:
       creds = None


   return creds, proxy_url


def process_request(self, request, spider):
   # ignore if proxy is already set
```

```
        if 'proxy' in request.meta:
            return

        parsed = urlparse_cached(request)
        scheme = parsed.scheme

        # 'no_proxy' is only supported by http schemes
        if scheme in ('http', 'https') and proxy_bypass
            return

        if scheme in self.proxies:
            self._set_proxy(request, scheme)

    def _set_proxy(self, request, scheme):
        creds, proxy = self.proxies[scheme]
        request.meta['proxy'] = proxy
        if creds:
            request.headers['Proxy-Authorization'] = b'
```

分析代码如下：

- _init_方法

  在HttpProxyMiddleware的构造器中，使用Python标准库urllib中的getproxies函数在系统环境变量中搜索系统代理的相关配置（变量名格式为[协议]_proxy的变量），调用self._get_proxy方法解析代理配置信息，并将其返回结果保存到self.proxies字典中，如果没有找到任何代理配置，就抛出NotConfigured异常，HttpProxyMiddleware被弃用。

- _get_proxy方法

  解析代理配置信息，返回身份验证信息（后面讲解）以及代理服务器url。

- process_request方法

处理每一个待发送的请求，为没有设置过代理的请求（meta属性不包含proxy字段的请求）调用self._set_proxy方法设置代理。

- _set_proxy方法

  为一个请求设置代理，以请求的协议（HTTP或HTTPS）作为键，从代理服务器信息字典self.proxies中选择代理，赋给request.meta的proxy字段。对于需要身份验证的代理服务器，添加HTTP头部Proxy-Authorization，其值是在_get_proxy方法中计算得到的。

经分析得知，在Scrapy中为一个请求设置代理的本质就是将代理服务器的url填写到request.meta['proxy']。

13.2 使用多个代理

利用HttpProxyMiddleware为爬虫设置代理时，对于一种协议（HTTP或HTTPS）的所有请求只能使用一个代理，如果想使用多个代理，可以在构造每一个Request对象时，通过meta参数的proxy字段手动设置代理：

```
request1 = Request('http://example.com/1', meta={'
request2 = Request('http://example.com/2', meta={'
request3 = Request('http://example.com/3', meta={'
```

按照与之前相同的做法，在scrapy shell进行实验，验证代理是否被使用：

```
$ scrapy shell
...
>>> from scrapy import Request
>>> req = Request('http://httpbin.org/ip', meta={'
>>> fetch(req)
```

```
[scrapy] DEBUG: Crawled (200) (referer: None)
>>> json.loads(response.text)
{'origin': '116.29.35.201'}
>>> req = Request('https://httpbin.org/ip', meta={
>>> fetch(req)
[scrapy] DEBUG: Crawled (200) (referer: None)
>>> json.loads(response.text)
{'origin': '197.10.171.143'}
```

结果表明，Scrapy爬虫同样使用了指定的代理服务器。

使用手动方式设置代理时，如果使用的代理需要身份验证，还需要通过HTTP头部的Proxy-Authorization字段传递包含用户账号和密码的身份验证信息。可以参考HttpProxyMiddleware._get_proxy中的相关实现，按以下过程生成身份验证信息：

（1）将账号、密码拼接成形如'user:passwd'的字符串s1。

（2）按代理服务器要求对s1进行编码（如utf8），生成s2。

（3）再对s2进行Base64编码，生成s3。

（4）将s3拼接到固定字节串b'Basic '后面，得到最终的身份验证信息。

示例代码如下：

```
>>> from scrapy import Request
>>> import base64
>>> req = Request('http://httpbin.org/ip', meta={'
>>> user = 'liushuo'
>>> passwd = '12345678'
>>> user_passwd = ('%s:%s' % (user, passwd)).encod
>>> user_passwd
b'liushuo:12345678'
>>> req.headers['Proxy-Authorization'] = b'Basic '
>>> fetch(req)
...
```

13.3 获取免费代理

现在，我们了解了如何为Scrapy爬虫设置代理，接下来的一个话题便是如何获取代理服务器。如果你感觉购买云服务器（亚马逊云或阿里云服务器等）自行搭建代理服务器的成本太高（但可靠、可控），那么可以通过google或baidu找到一些提供免费代理服务器信息的网站，例如：

http://proxy-list.org（国外）

https://free-proxy-list.net（国外）

http://www.xicidaili.com

http://www.proxy360.cn

http://www.kuaidaili.com

以http://www.xicidaili.com为例，图13-3所示为该网站“国内高匿代理”分类下的页面。

www.xicidaili.com/nn/1

国内高匿代理IP
XiciDaili.com

首页 API提取 国内高匿代理 国内普通代理 国内HTTPS代理 国内HTTP代理 代理小百科

当前位置：首页 > 国内高匿代理

国家	IP地址	端口	服务器地址	是否匿名	类型	速度	连接时间	存活时间	验证时间
	114.239.148.175	808	江苏宿迁市泗阳县	高匿	HTTP			43天	17-04-27 15:21
	221.216.94.77	808	北京	高匿	HTTP			44天	17-04-27 15:21
	61.191.173.31	808	安徽淮南	高匿	HTTP			29天	17-04-27 15:21
	171.38.166.94	8123	广西钦州	高匿	HTTP			263天	17-04-27 15:20
	61.143.158.238	808	广东揭阳	高匿	HTTP			274天	17-04-27 15:18
	182.88.106.161	8123	广西南宁	高匿	HTTPS			12小时	17-04-27 15:17
	110.73.3.243	8123	广西防城港	高匿	HTTP			175天	17-04-27 15:15
	122.5.132.240	808	山东德州	高匿	HTTP			1分钟	17-04-27 15:15
	171.38.165.48	8123	广西钦州	高匿	HTTPS			4小时	17-04-27 15:15
	218.22.219.133	808	安徽黄山	高匿	HTTP			29天	17-04-27 15:13
	175.155.24.119	808	四川德阳	高匿	HTTP			56天	17-04-27 15:11
	59.62.99.170	808	江西鹰潭	高匿	HTTPS			1分钟	17-04-27 15:11
	221.229.47.218	808	江苏扬州	高匿	HTTPS			2小时	17-04-27 15:11
	120.83.100.120	808	广东揭阳市普宁	高匿	HTTPS			1分钟	17-04-27 15:11
	117.43.0.212	808	江西九江	高匿	HTTPS			1分钟	17-04-27 15:11
	115.214.10.108	8118	浙江宁波	高匿	HTTP			2小时	17-04-27 15:11
	171.13.36.72	808	河南	高匿	HTTP			29天	17-04-27 15:11
	116.28.104.156	808	广东中山	高匿	HTTPS			1分钟	17-04-27 15:11

图13-3

从中可以看出，该网站提供了大量的免费代理服务器信息，如果只需要少量的代理，从中选择几个就可以了，不过通常直觉告诉我们“多多益善”。接下来编写爬虫，爬取“国内高匿代理”分类下前3页的所有代理服务器信息，并验证每个代理是否可用。

创建Scrapy项目，取名为proxy_example：

```
$ scrapy startproject proxy_example
```

该网站会检测用户发送的HTTP请求头部中的User-Agent字段，因此我们需要伪装成某种常规浏览器，在配置文件添加如下代码：

```
USER_AGENT = 'Mozilla/5.0 (X11; Linux x86_64) Appl
```

实现XiciSpider爬取代理服务器信息，并过滤不可用代理，代码如下：

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy import Request


import json


class XiciSpider(scrapy.Spider):
   name = "xici_proxy"
   allowed_domains = ["www.xicidaili.com"]
def start_requests(self):
   #爬取http://www.xicidaili.com/nn/前3 页
   foriin range(1, 4):
       yield Request('http://www.xicidaili.com/nn/


def parse(self, response):
   for sel in response.xpath('//table[@id="ip_list
       # 提取代理的IP、port、scheme(http or https)
       ip = sel.css('td:nth-child(2)::text').extra
       port  = sel.css('td:nth-child(3)::text').ex
       scheme = sel.css('td:nth-child(6)::text').e
```

```
        # 使用爬取到的代理再次发送请求到http(s)://httpbi
        url = '%s://httpbin.org/ip' % scheme
        proxy = '%s://%s:%s' % (scheme, ip, port)

        meta = {
            'proxy': proxy,
            'dont_retry': True,
            'download_timeout': 10,

            # 以下两个字段是传递给check_available 方法的
            '_proxy_scheme': scheme,
            '_proxy_ip': ip,
        }

        yield Request(url, callback=self.check_avai
                            meta=meta, dont_filter

def check_available(self, response):
    proxy_ip = response.meta['_proxy_ip']
    # 判断代理是否具有隐藏IP 功能
    if proxy_ip == json.loads(response.text)['origi
        yield {
            'proxy_scheme': response.meta['_proxy_sc
            'proxy': response.meta['proxy'],
        }
```

解释上述代码如下：

- 在start_requests方法中请求http://www.xicidaili.com/nn下的前3页，以parse方法作为页面解析函数。
- 在parse方法中提取一个页面中所有的代理服务器信息，这些代理未必都是可用的，因此使用爬取到的代理发送请求到http(s)://httpbin.org/ip验证其是否可用，以check_available方法作为页面解析函数。

- 能执行到check_available方法，意味着response对应请求所使用的代理是可用的。在check_available方法中，通过响应json串中的origin字段可以判断代理是否是匿名的（隐藏ip），返回匿名代理。

运行爬虫，将可用的代理服务器保存到json文件中，供其他程序使用：

```
$ scrapy crawl xici_proxy -o proxy_list.json
...
$ cat proxy_list.json
[
{"proxy": "http://110.73.10.37:8123", "proxy_schem
{"proxy": "http://171.38.142.24:8123", "proxy_sche
{"proxy": "http://111.155.124.84:8123", "proxy_sch
{"proxy": "http://203.88.210.121:138", "proxy_sche
{"proxy": "http://182.88.191.195:8123", "proxy_sch
{"proxy": "http://121.31.151.231:8123", "proxy_sch
{"proxy": "http://203.93.0.115:80", "proxy_scheme"
{"proxy": "http://222.85.39.29:808", "proxy_scheme
{"proxy": "http://175.155.25.26:808", "proxy_schem
{"proxy": "http://111.155.124.72:8123", "proxy_sch
{"proxy": "http://122.5.81.153:8118", "proxy_schem
{"proxy": "https://171.37.153.24:8123", "proxy_sch
{"proxy": "http://110.73.1.68:8123", "proxy_scheme
{"proxy": "http://122.228.179.178:80", "proxy_sche
{"proxy": "http://121.31.151.226:8123", "proxy_sch
{"proxy": "http://171.38.171.168:8123", "proxy_sch
{"proxy": "http://218.22.219.133:808", "proxy_sche
{"proxy": "https://171.38.130.188:8123", "proxy_sc
{"proxy": "http://111.155.116.219:8123", "proxy_sc
{"proxy": "http://121.31.149.209:8123", "proxy_sch
{"proxy": "http://60.169.78.218:808", "proxy_schem
{"proxy": "http://171.38.158.227:8123", "proxy_sch
{"proxy": "http://121.31.150.224:8123", "proxy_sch
{"proxy": "http://111.155.124.78:8123", "proxy_sch
{"proxy": "http://182.90.83.104:8123", "proxy_sche
]
```

如结果所示，我们成功获取到了20多个可用的免费代理。

13.4 实现随机代理

本章开始部分曾提到，某些网站为防止爬虫爬取会对接收到的请求进行监测，如果在短时间内接收到了来自同一IP的大量请求，就判定该IP的主机在使用爬虫程序爬取网站，因而将该IP封禁（拒绝请求）。爬虫程序可以使用多个代理对此类网站进行爬取，此时单位时间的访问量会被多个代理分摊，从而避免封禁IP。

下面我们基于HttpProxyMiddleware实现一个随机代理下载中间件。

在middlewares.py中实现RandomHttpProxyMiddleware，代码如下：

```
from scrapy.downloadermiddlewares.httpproxy import
from collections import defaultdict
import json
import random


class RandomHttpProxyMiddleware(HttpProxyMiddlewar


    def __init__(self, auth_encoding='latin-1', pro
        if not proxy_list_file:
            raise NotConfigured


        self.auth_encoding = auth_encoding
        # 分别用两个列表维护HTTP和HTTPS的代理，{'http':
        self.proxies = defaultdict(list)


        # 从json文件中读取代理服务器信息，填入self.proxi
```

```
        with open(proxy_list_file) as f:
            proxy_list = json.load(f)
            for proxy in proxy_list:
                scheme = proxy['proxy_scheme']
                url = proxy['proxy']
            self.proxies[scheme].append(self._get_pr


    @classmethod
    def from_crawler(cls, crawler):
        # 从配置文件中读取用户验证信息的编码
        auth_encoding = crawler.settings.get('HTTPPROXY


        # 从配置文件中读取代理服务器列表文件（json）的路径
        proxy_list_file = crawler.settings.get('HTTPPRO


        return cls(auth_encoding, proxy_list_file)


    def _set_proxy(self, request, scheme):
        # 随机选择一个代理
        creds, proxy = random.choice(self.proxies[schem
        request.meta['proxy'] = proxy
        if creds:
            request.headers['Proxy-Authorization'] = b'
```

解释上述代码如下：

- 仿照HttpProxyMiddleware构造器实现RandomHttpProxyMiddleware构造器，首先从代理服务器列表文件（配置文件中指定）中读取代理服务器信息，然后将它们按协议（HTTP或HTTPS）分别存入不同列表，由self.proxis字典维护。
- _set_proxy方法负责为每一个Request请求设置代理，覆写_set_proxy方法（覆盖基类方法）。对于每一个request，根据请求协议获取self.proxis中的代理服务器列表，然后从中

随机抽取一个代理，赋值给request.meta['proxy']。

在配置文件settings.py中启用RandomHttpProxyMiddleware，并指定所要使用的代理服务器列表文件（json文件），添加代码如下：

```
DOWNLOADER_MIDDLEWARES = {
    # 置于HttpProxyMiddleware(750)之前
    'proxy_example.middlewares.RandomHttpProxyMiddl
}


# 使用之前在http://www.xicidaili.com/网站爬取到的代理
HTTPPROXY_PROXY_LIST_FILE='proxy_list.json'
```

最后编写一个TestRandomProxySpider测试该中间件，重复向http(s)://httpbin.org/ip发送请求，根据响应中的请求源IP地址信息判断代理使用情况：

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy import Request
import json


class TestRandomProxySpider(scrapy.Spider):
    name = "test_random_proxy"


    def start_requests(self):
        for _ in range(100):
            yield Request('http://httpbin.org/ip', dc
            yield Request('https://httpbin.org/ip', d


    def parse(self, response):
        print(json.loads(response.text))
```

运行爬虫，观察输出：

```
$ scrapy crawl test_random_proxy
[scrapy] DEBUG: Crawled (200) <GET https://httpbin
{'origin': '171.38.130.188'}
[scrapy] DEBUG: Crawled (200) <GET http://httpbin.
{'origin': '182.90.83.104'}
[scrapy] DEBUG: Crawled (200) <GET https://httpbin
{'origin': '171.38.130.188'}
[scrapy] DEBUG: Crawled (200) <GET http://httpbin.
{'origin': '203.88.210.121'}
[scrapy] DEBUG: Crawled (200) <GET http://httpbin.
{'origin': '203.88.210.121'}
...
```

结果表明，RandomHttpProxyMiddleware工作良好，Scrapy爬虫随机地使用了多个代理。

13.5　项目实战：爬取豆瓣电影信息

最后，我们来完成一个使用代理爬取的实战项目。豆瓣网的电影专栏是国内权威电影评分网站，其中包括海量影片信息，在浏览器中访问https://movie.douban.com，并选择一个分类（如“豆瓣高分”），可看到如图13-4所示的影片列表页面。

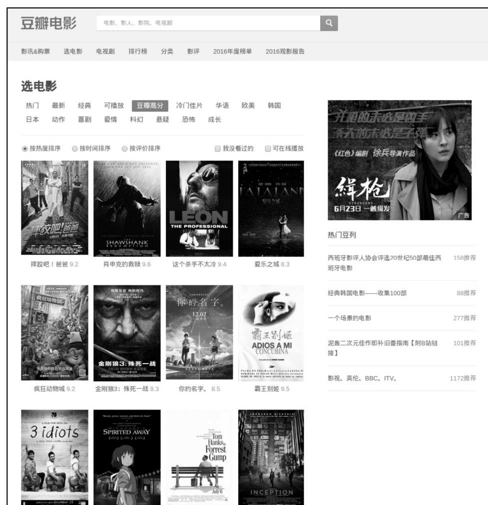


图13-4

单击其中一部影片，进入其页面（简称影片页面），如图13-5所示。

在影片页面中可以看到一部影片的基本信息，如导演、编剧、主演、类型等，我们可以编写爬虫在豆瓣电影中爬取大量影片信息。



图13-5

13.5.1　项目需求

爬取豆瓣电影中“豆瓣高分”分类下的所有影片信息，需要爬取一部影片的信息字段如下：

- 导演
- 编剧
- 主演
- 类型
- 制片国家 / 地区
- 语言
- 上映日期
- 片长
- 又名

由于豆瓣网对爬取速度做了限制，高速爬取可能会被封禁IP，因此使用代理进行爬取。

13.5.2　页面分析

首先分析影片列表页面，页面中的每一部电影都是通过JavaScript脚本加载的。单击页面最下方的“加载更多”，可以在Chrome开发者工具中捕获到jQuery发送的HTTP请求（加载更多影片），该请求返回了一个json串，如图13-6所示。

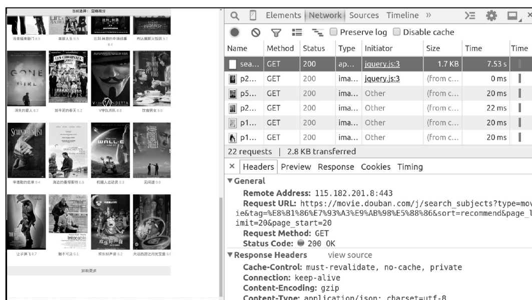


图13-6

复制图中请求的url，使用scrapy shell进行访问，查看其中json串的内容：

```
    $ scrapy shell
'https://movie.douban.com/j/search_subjects?type=m
%E5%88%86&sort=recommend&page_limit=20&page_start=
    ...
>>> import json
>>> res = json.loads(response.body.decode('utf8'))
>>> res
{'subjects': [
 {'cover': 'https://img1.doubanio.com/view/movie_p
  'cover_x': 1538,
  'cover_y': 2159,
  'id': '1292001',
  'is_beetle_subject': False,
  'is_new': False,
  'playable': False,
  'rate': '9.2',
  'title': '海上钢琴师',
  'url': 'https://movie.douban.com/subject/1292001
  {'cover': 'https://img1.doubanio.com/view/movie_
  'cover_x': 1500,
  'cover_y': 2145,
  'id': '25986180',
  'is_beetle_subject': False,
  'is_new': False,
```

```
'playable': False,
'rate': '8.2',
'title': '釜山行',
'url': 'https://movie.douban.com/subject/2598618
{'cover': 'https://img1.doubanio.com/view/movie_
'cover_x': 703,
'cover_y': 1000,
'id': '26580232',
'is_beetle_subject': False,
'is_new': False,
'playable': False,
'rate': '8.7',
'title': '看不见的客人',
'url': 'https://movie.douban.com/subject/2658023

...省略中间部分...

{'cover': 'https://img3.doubanio.com/view/movie_
'cover_x': 2181,
'cover_y': 3120,
'id': '6786002',
'is_beetle_subject': False,
'is_new': False,
'playable': False,
'rate': '9.1',
'title': '触不可及',
'url': 'https://movie.douban.com/subject/6786002
{'cover': 'https://img3.doubanio.com/view/movie_
'cover_x': 1000,
'cover_y': 1500,
'id': '26354572',
'is_beetle_subject': False,
'is_new': False,
'playable': True,
'rate': '8.2',
'title': '欢乐好声音',
'url': 'https://movie.douban.com/subject/2635457
{'cover': 'https://img3.doubanio.com/view/movie_
```

```
    'cover_x': 1005,
    'cover_y': 1437,
    'id': '1299398',
    'is_beetle_subject': False,
    'is_new': False,
    'playable': False,
    'rate': '8.9',
    'title': '大话西游之月光宝盒',
    'url': 'https://movie.douban.com/subject/1299398
  ]}
```

如上所示，返回结果（json）中的'subjects'字段是一个影片信息列表，一共有20项，每一项都是一部影片的信息，其中可以找到影片片名（title）、评分（rate）以及影片页面url等信息。

连续单击加载按钮，捕获更多jQuery发送的HTTP请求，可以总结出其url的规律：

- type参数：类型，movie代表电影。
- tag参数：分类标签，当前为“豆瓣高分”。
- page_start：从第几部影片开始加载，即结果列表中第一部影片在服务器端的序号。
- page_limit：期望获取的影片信息的数量，当前为20。

我们可以通过分析出的API，每次获取固定数量的影片信息，从中提取每一个影片页面的url，例如：

先获取20部影片信息：

```
[BASE_URL]?type=movie&tag=豆瓣高分&sort=recommend&p
```

再获取20部影片信息：

```
[BASE_URL]?type=movie&tag=豆瓣高分&sort=recommend&p
```

再获取20部影片信息：

```
[BASE_URL]?type=movie&tag=豆瓣高分&sort=recommend&p
```

…

直到返回结果中的影片信息列表为空，说明没有影片了。

接下来分析影片页面。在scrapy shell中下载任意一个影片页面，并调用view函数在浏览器中查看页面，如图13-7所示。



图13-7

从图13-7中可以看出，影片的信息在<div id="info">中，其中每一个信息字段名（"导演"，"编剧"等）都位于一个<span class="pl">中，比较容易提取，但字段的值很难找到统一的规律，我们可以使用XPath的string函数将<div id="info">中的所有文本提取到一个字符串，然后用提取到的字段名分割该字符串，得到其中值的部分。

首先提取包含所有信息的字符串：

```
>>> info = response.css('div#info').xpath('str
>>> print(info)
导演：涅提·蒂瓦里
编剧：比于什·古普塔 / 施热亚·简
```

```
    主演：阿米尔·汗 / 法缇玛·萨那·纱卡 / 桑亚·玛荷塔 / 阿
沃
    / 泽伊拉·沃西姆 / 苏哈妮·巴特纳格尔 / 里特维克·萨霍里
    类型：剧情 / 传记 / 运动
    制片国家/地区：印度
   语言：印地语
    上映日期：2017-05-05(中国大陆) / 2016-12-23(印度)
    片长：161分钟(印度) / 140 分钟(中国大陆)
    又名：我和我的冠军女儿(台) / 打死不离3 父女(港) / 摔跤
    Wrestling Competition
    IMDb链接: tt5074352
```

再提取所有字段名到一个列表：

```
>>> fields = [s.strip().replace(':', '') for s in
...             response.css('div#info span.pl::te
>>> fields
['导演','编剧','主演','类型','制片国家/地区','语言','上
```

使用字段名对info进行分割，得到所有值的列表：

```
>>> import re
>>> values = [re.sub('\s+', ' ', s.strip()) for s
...           re.split('\s*(?:%s):\s*' % '|'.join(
>>> values
['涅提·蒂瓦里',
'比于什·古普塔 / 施热亚·简',
'阿米尔·汗 / 法缇玛·萨那·纱卡 / 桑亚·玛荷塔 / 阿帕尔夏克拉
泽伊拉·沃西姆 / 苏哈妮·巴特纳格尔 / 里特维克·萨霍里 / 吉里
 '剧情 / 传记 / 运动',
 '印度',
 '印地语',
 '2017-05-05(中国大陆) / 2016-12-23(印度)',
 '161 分钟(印度) / 140 分钟(中国大陆)',
 '我和我的冠军女儿(台) / 打死不离3 父女(港) / 摔跤吧！老爸
Wrestling Competition',
```

```
    'tt5074352']
```

最后，使用以上两个列表构造影片信息字典：

```
        >>> dict(zip(fields, values))
        {'IMDb链接': 'tt5074352',
         '上映日期': '2017-05-05(中国大陆) / 2016-12-23
         '主演':'阿米尔·汗 / 法缇玛·萨那·纱卡 / 桑亚·玛荷塔
    希·坦沃 /
        泽伊拉·沃西姆 / 苏哈妮·巴特纳格尔 / 里特维克·萨霍里
         '制片国家/地区': '印度',
         '又名':'我和我的冠军女儿(台) / 打死不离3 父女(港)
        Wrestling Competition',
         '导演':'涅提·蒂瓦里',
         '片长': '161 分钟(印度) / 140 分钟(中国大陆)',
         '类型':'剧情 / 传记 / 运动',
         '编剧': '比于什·古普塔 / 施热亚·简',
         '语言':'印地语'}
```

经上述操作，我们得到了除片名和评分之外的所有信息，片名和评分信息可以在json串中获取。

到此，页面分析的工作完成了。

13.5.3　编码实现

创建Scrapy项目，取名为douban_movie：

```
$ scrapy startproject douban_movie
```

在页面分析中，我们详细阐述了爬取过程，现在可以轻松实现MoviesSpider了，代码如下：

```
# -*- coding: utf-8 -*-
```

```
import scrapy
      from scrapy import Request
      import json
      import re
      from pprint import pprint


      class MoviesSpider(scrapy.Spider):
          BASE_URL = 'https://movie.douban.com/j/se
recommend&page_limit=%s&page_start=%s'
          MOVIE_TAG      = '豆瓣高分'
          PAGE_LIMIT = 20
          page_start = 0


          name = "movies"
          start_urls = [BASE_URL % (MOVIE_TAG, PAGE


          def parse(self, response):
              # 使用json 模块解析响应结果
              infos = json.loads(response.body.decod


              # 迭代影片信息列表
              for movie_info in infos['subjects']:
                  movie_item = {}


                  # 提取"片名"和"评分", 填入item.
                  movie_item['片名'] = movie_info['ti
                  movie_item['评分'] = movie_info['ra


                  # 提取影片页面url，构造Request 发送请求
页面解析函数
                  yield Request(movie_info['url'], c
                                                   meta=
```

```
                # 如果json 结果中包含的影片数量小于请求数量，
                if len(infos['subjects']) == self.PAGE
                    self.page_start += self.PAGE_LIMIT
                    url = self.BASE_URL % (self.MOVIE_
                    yield Request(url)
    def parse_movie(self, response):
        # 从meta 中提取已包含"片名"和"评分"信息的item
        movie_item = response.meta['_movie_item']


        # 获取整个信息字符串
        info = response.css('div.subject div#info').xpa


        # 提取所有字段名
        fields= [s.strip().replace(':', '') for s in \
                      response.css('div#info span.pl::


        # 提取所有字段的值
        values = [re.sub('\s+', ' ', s.strip()) for s i
                      re.split('\s*(?:%s):\s*' % '|'.j


        # 将所有信息填入item
        movie_item.update(dict(zip(fields, values)))


        yield movie_item
```

为了使用代理进行爬取，我们将之前实现的RandomHttpProxyMiddleware复制到该项目中。

在配置文件settings.py中添加如下配置：

```
    # 我们的爬取不符合豆瓣爬取规则，强制爬取
    ROBOTSTXT_OBEY = False
```

```
# 伪装成常规浏览器
USER_AGENT = 'Mozilla/5.0 (X11; Linux x86_64) Chro

# 可选。设置下载延迟，防止代理被封禁IP，依据代理数量而定
DOWNLOAD_DELAY = 0.5

# 启用随机代理中间件
DOWNLOADER_MIDDLEWARES = {
    'douban_movie.middlewares.RandomHttpProxyMiddle
}

# 指定所要使用的代理服务器列表文件
HTTPPROXY_PROXY_LIST_FILE = 'proxy_list.json'
```

为了爬取稳定，使用在云服务器上自行搭建的代理服务器，手工编辑代理服务器列表文件proxy_list.json：

```
[
{"proxy_scheme": "https", "proxy": "http://116.29.
{"proxy_scheme": "https", "proxy": "http://197.10.
{"proxy_scheme": "https", "proxy": "http://112.78.
{"proxy_scheme": "https", "proxy": "http://124.59.
]
```

最后，运行爬虫，将结果保存到文件moveis.json：

```
$ scrapy crawl movies -o movies.json
```

在Python中观察爬取结果，代码如下：

```
>>> import json
>>> movies = json.load(open('movies.json'))
>>> for movie in movies:
...     print(movie['片名'], movie['评分'], movie['导
...
忠犬八公的故事 9.2 拉斯·霍尔斯道姆
楚门的世界 9.0 彼得·威尔
怦然心动 8.9 罗伯·莱纳
泰坦尼克号 9.2 詹姆斯·卡梅隆
血战钢锯岭 8.7 梅尔·吉布森
驴得水 8.3 周申 / 刘露
星际穿越 9.1 克里斯托弗·诺兰
盗梦空间 9.2 克里斯托弗·诺兰
阿甘正传 9.4 罗伯特·泽米吉斯
千与千寻 9.2 宫崎骏
三傻大闹宝莱坞 9.1 拉吉库马尔·希拉尼
霸王别姬 9.5 陈凯歌
你的名字 8.5 新海诚
金刚狼3：殊死一战 8.3 詹姆斯·曼高德
疯狂动物城 9.2 拜伦·霍华德 / 瑞奇·摩尔 / 杰拉德·布什
爱乐之城 8.3 达米恩·查泽雷
大话西游之月光宝盒 8.9 刘镇伟
触不可及 9.1 奥利维·那卡什 / 艾力克·托兰达
无间道 9.0 刘伟强 / 麦兆辉
让子弹飞 8.7 姜文
...省略中间部分...
哈尔的移动城堡 8.8 宫崎骏
阿凡达 8.6 詹姆斯·卡梅隆
教父 9.2 弗朗西斯·福特·科波拉
罗马假日 8.9 威廉·惠勒
龙猫 9.1 宫崎骏
火星救援 8.4 雷德利·斯科特
超能陆战队 8.6 唐·霍尔 / 克里斯·威廉姆斯
欢乐好声音 8.2 加斯·詹宁斯 / 克里斯托夫·卢尔德莱
少年派的奇幻漂流 9.0 李安
釜山行 8.2 延尚昊
大话西游之大圣娶亲 9.2 刘镇伟
这个杀手不太冷 9.4 吕克·贝松
海上钢琴师 9.2 朱塞佩·托纳多雷
宣告黎明的露之歌 8.1 汤浅政明
```

```
肖申克的救赎 9.6 弗兰克·德拉邦特
摔跤吧！爸爸 9.2 涅提·蒂瓦里
>>> len(movies)
500
```

如上所示，我们成功爬取了“豆瓣高分”分类下的500部影片信息。

13.6 本章小结

本章学习了Scrapy爬虫如何使用代理进行爬取，首先介绍了两种设置代理的方法：

- 使用下载中间件HttpProxyMiddleware（自动）。
- 在构造Request对象时通过meta参数设置（手动）。

前者使用简单，只需通过环境变量配置即可；后者可在某些特殊场景下使用。我们以如何使用多个代理进行了举例，随后讲解了在网络上获取免费代理的方法，并利用获取的免费代理实现了一个随机代理中间件。最后，运用本章所学的知识完成一个实战项目，使用代理爬取了豆瓣网中的电影信息。

第14章 分布式爬取

由于受到计算能力和网络带宽的限制，单台计算机上运行的爬虫在爬取的数据量较大时，需要耗费很长的时间。分布式爬取的思想是“人多力量大”，在网络中的多台计算机上同时运行爬虫程序，共同完成一个大型爬取任务。这一章来学习使用Scrapy框架进行分布式爬取。

Scrapy本身并不是一个为分布式爬取而设计的框架，但第三方库scrapy-redis为其拓展了分布式爬取的功能，两者结合便是一个分布式Scrapy爬虫框架。在分布式爬虫框架中，需要使用某种通信机制协调各个爬虫的工作，让每一个爬虫明确自己的任务，其中包括：

（1）当前的爬取任务，即下载＋提取数据（分配任务）。

（2）当前爬取任务是否已经被其他爬虫执行过（任务去重）。

（3）如何存储爬取到的数据（数据存储）。

scrapy-redis利用Redis数据库作为多个爬虫的数据共享实现以上功能，接下来我们学习如何使用scrapy-redis进行分布式爬取。

14.1　Redis的使用

首先来学习Redis数据库的使用。Redis是一个速度非常快的非关系型数据库，使用内存作为主存储，内存中的数据也可以被持久化到硬盘。Redis以键值形式（key-value）存储数据，其中的值可以分为以下5种类型：

- 字符串（string）
- 哈希（hash）
- 列表（list ）

- 集合（set）
- 有序集合（zset）

14.1.1　安装Redis

接下来安装Redis，在Ubuntu下可以使用apt-get安装：

```
sudo apt-get install redis-server
```

Redis数据库进程是一个网络服务器，可以使用以下命令开启/重启/停止Redis：

```
$ sudo service redis-server start   # 开启Redis
$ sudo service redis-server restart # 重启Redis
$ sudo service redis-server start   # 停止Redis
```

默认情况下，Redis会在127.0.0.1:6379上开启服务，可以使用netstat命令进行查询：

```
$ netstat -ntl
Proto Recv-Q Send-Q Local Address     Foreign Addre
...
tcp        0      0 127.0.0.1:6379    *:*
...
```

此时，我们运行的Redis仅能被本机使用，因为它只接收来自本机（localhost）的请求。如果想让Redis能被其他计算机访问，需要在配置文件中修改Redis服务器的监听地址，在Redis配置文件/etc/redis/redis.conf中做以下修改：

```
$ sudo vi /etc/redis/redis.conf
```

```
...
# bind 127.0.0.1
bind 0.0.0.0  # 接收来自任意IP的请求
...
```

重启Redis后，修改生效：

```
$ sudo service redis-server restart
Stopping redis-server: redis-server.
Starting redis-server: redis-server.
$ netstat -ntl
Proto Recv-Q Send-Q Local Address     Foreign Addr
...
$ tcp        0       0 0.0.0.0:6379      0.0.0.0:*
...
```

现在，可以在任意机器上使用客户端连接Redis数据库：

```
$ redis-cli -h 192.168.0.103  # 使用-h 参数指定主机IP
192.168.1.102:6379> PING      # 测试链接数据库是否成功
PONG
```

14.1.2 Redis基本命令

由于篇幅有限，这里仅介绍一些Redis基本使用命令，按照值的5种类型依次讲解。想学习Redis更多详细内容，请参考相关书籍或Redis官方网站https://redis.io。

字符串

Redis的字符串（String）可以存储字符串、整数、浮点数（数字也是字符串）。String命令及描述如表14-1所示。

表**14-1 String**命令及描述

String 命令	描　述
SET key value	设置字符串 key 的值
GET key	获取字符串 key 的值
DEL key	删除 key

```
192.168.1.102:6379> SET x abcde
OK
192.168.1.102:6379> GET x
"abcde"
192.168.1.102:6379> DEL x
(integer) 1
192.168.1.102:6379> GET x
(nil)
```

列表

Redis的列表（List）可以有序地存储多个字符串。List命令及描述如表14-2所示。

表14-2　List命令及描述

List 命令	描　述
LPUSH key value1 [value2...]	在列表 key 左端插入一个或多个值
RPUSH key value1 [value2...]	在列表 key 右端插入一个或多个值
LPOP key	从列表 key 左端弹出一个值
RPOP key	从列表 key 右端弹出一个值
LINDEX key index	获取列表 key 中 index 位置的值
LRANGE key start end	获取列表 key 中位置在[start, end]范围的值
LLEN key	获取列表 key 的长度

```
192.168.1.102:6379> RPUSH color orange              # 右边
(integer) 1
192.168.1.102:6379> RPUSH color red yellow blue  # 右边
(integer) 4
192.168.1.102:6379> LINDEX color 2                  # 获取
"yellow"
192.168.1.102:6379> LRANGE color 0 3                # 获取
1) "orange"
2) "red"
```

```
3) "yellow"
4) "blue"
192.168.1.102:6379> LPOP color                              # 左边出
"orange"
192.168.1.102:6379> LRANGE color 0 -1                       # 获取全
1) "red"
2) "yellow"
3) "blue"
```

哈希

Redis的哈希（Hash）可以存储多个键值对，其中的键和值都是字符串。Hash命令及描述如表14-3所示。

表14-3 Hash命令与描述

Hash 命令	描　述
HSET key field value	将哈希 key 的 field 字段赋值为 value
HDEL key field1 [field2...]	删除哈希 key 的一个或多个字段
HGET key field	获取哈希 key 的 field 字段的值
HGETALL key	获取哈希 key 的所有字段和值

```
192.168.1.102:6379> HSET point x 34   # 赋值x、y、z 三个字
(integer) 1
192.168.1.102:6379> HSET point y 55
(integer) 1
192.168.1.102:6379> HSET point z 47
(integer) 1
192.168.1.102:6379> HGET point x        # 获取字段x的值
"34"
192.168.1.102:6379> HGETALL point        # 获取所有字段和值
1) "x"
2) "34"
3) "y"
4) "55"
5) "z"
6) "47"
192.168.1.102:6379> HDEL point y        # 删除字段y
(integer) 1
192.168.1.102:6379> HGETALL point
```

```
1) "x"
2) "34"
3) "z"
4) "47"
```

集合

Redis中的集合（Set）可以存储多个唯一的字符串。Set命令及描述如表14-4所示。

表14-4　Set命令及描述

Set 命令	描　　述
SADD key member1 [member2...]	向集合 key 中添加一个或多个成员
SREM key member1 [member2...]	删除集合 key 中一个或多个成员
SMEMBERS key	获取集合 key 中所有成员
SCARD key	获取集合 key 中成员数量
SISMEMBER key member	判断 member 是否是集合 key 的成员

```
192.168.1.102:6379> SADD color_set white black red blue
(integer) 4
192.168.1.102:6379> SMEMBERS color_set
1) "blue"
2) "red"
3) "black"
4) "white"
192.168.1.102:6379> SADD color_set white
(integer) 0
192.168.1.102:6379> SCARD color_set
(integer) 4
192.168.1.102:6379> SREM color_set red
(integer) 1
192.168.1.102:6379> SMEMBERS color_set
1) "black"
2) "white"
3) "blue"
192.168.1.102:6379> SISMEMBER color_set blue
(integer) 1
192.168.1.102:6379> SISMEMBER color_set red
(integer) 0
```

有序集合

Redis中的有序集合（ZSet）与集合（Set）类似，可以存储多个唯一的字符串，但在有序集合中，每个成员都有一个分数，所有成员按给定分数在集合中有序排列。Zset命令及描述如表14-5所示。

表14-5　ZSet命令及描述

ZSet 命令	描　　述
ZADD key score1 member1 [score2 member2...]	向有序集合 key 中添加一个或多个成员
ZREM key member [member2 ...]	删除有序集合 key 中一个或多个成员
ZRANGE key start stop	获取有序集合 key 中位置在[start, stop]范围的所有成员
ZRANGEBYSCORE key min max	获取有序集合 key 中分值在[start, stop]范围的所有成员

```
192.168.1.102:6379> ZADD language 3 python 6 perl 9 lis
(integer) 3
192.168.1.102:6379> ZADD language 1 C 4 java 5 C++
(integer) 3
192.168.1.102:6379> ZREM language C++
(integer) 1
192.168.1.102:6379> ZRANGE language 2 4
1) "java"
2) "perl"
3) "lisp"
192.168.1.102:6379> ZRANGEBYSCORE language 3 6
1) "python"
2) "java"
3) "perl"
```

14.1.3　Python访问Redis

Redis支持多种语言API，在Python中可以使用第三方库redis-py访问Redis数据库。

使用pip安装redis-py：

```
$ sudo pip install redis
```

redis-py的使用非常简单，只需先连接Redis获得连接对象，然后调用该对象上与每一条Redis命令相对应的方法即可。请看下面的简单示例：

```
>>> import redis
>>> r = redis.StrictRedis(host='localhost', port=6
>>> r.set('s', 'hello world')
True
>>> r.get('s')
b'hello world'
>>> r.rpush('queue', 1, 2, 3)  # 列表
3
>>> r.lpop('queue')
b'1'
>>> r.llen('queue')
2
>>> r.lrange('queue', 0, -1)
[b'2', b'3']
```

Redis简介部分到此结束了，希望这部分内容能够帮助大家读懂scrapy-redis的源码。

14.2　scrapy-redis源码分析

在使用scrapy-redis前，我们先来分析scrapy-redis的源码，了解其实现原理。

使用git从github网站下载scrapy-redis源码：

```
$ git clone https://github.com/rolando/scrapy-redis
$ cd scrapy-redis/src/
$ tree
.
 └─── scrapy_redis
      ├─── connection.py
      ├─── defaults.py
      ├─── dupefilter.py
      ├─── __init__.py
      ├─── picklecompat.py
      ├─── pipelines.py
      ├─── queue.py
      ├─── scheduler.py
      ├─── spiders.py
      └─── utils.py
1 directory, 10 files
```

scrapy-redis的源码并不多，因为它仅是利用Redis数据库重新实现了Scrapy中的某些组件。

对于一个分布式爬虫框架，需要解决以下两个最基本的问题。

- 分配爬取任务：为每个爬虫分配不重复的爬取任务。
- 汇总爬取数据：将所有爬虫爬取到的数据汇总到一处。

接下来我们看scrapy-redis是如何解决的。

14.2.1 分配爬取任务部分

scrapy-redis为多个爬虫分配爬取任务的方式是：让所有爬虫共享一个存在于Redis数据库中的请求队列（替代各爬虫独立的请求队列），每个爬虫从请求队列中获取请求，下载并解析页面后，将解析出的新请求再添加到请求队列中，因此每个爬虫既是下载任务的生产者又是消费者。

为实现多个爬虫的任务分配，scrapy-redis重新实现了以下组件：

- 基于Redis的请求队列（优先队列、FIFO、LIFO）。
- 基于Redis的请求去重过滤器（过滤掉重复的请求）。
- 基于以上两个组件的调度器。

1．调度器的实现

首先来看调度器Scheduler的实现，它位于scheduler.py中，Scheduler的核心代码如下：

```
import importlib
import six


from scrapy.utils.misc import load_object
from . import connection, defaults


class Scheduler(object):
    ...
    def open(self, spider):
        ...
        # 初始化请求队列
        try:
            self.queue = load_object(self.queue_cls)(
                server=self.server,
                spider=spider,
                key=self.queue_key % {'spider': spider
              serializer=self.serializer,
        )
    except TypeError as e:
        raise ValueError("Failed to instantiate que
                         self.queue_cls, e)


    # 初始化去重过滤器
    try:
        self.df = load_object(self.dupefilter_cls)(
            server=self.server,
            key=self.dupefilter_key % {'spider': spi
            debug=spider.settings.getbool('DUPEFILTE
        )
    except TypeError as e:
        raise ValueError("Failed to instantiate dup
                         self.dupefilter_cls, e)
```

```
        ...


    ...
    def enqueue_request(self, request):
        # 调用去重过滤器的request_seen 方法, 判断该request 
        # 如果页面已经爬取过，且用户没有强制忽略过滤，就直接返回
        if not request.dont_filter and self.df.request_
            self.df.log(request, self.spider)
            return False
        if self.stats:
            self.stats.inc_value('scheduler/enqueued/re
        # 将request 入队到请求队列
        self.queue.push(request)
        return True


    def next_request(self):
        block_pop_timeout = self.idle_before_close
        # 从请求队列出队一个request
        request = self.queue.pop(block_pop_timeout)
        if request and self.stats:
            self.stats.inc_value('scheduler/dequeued/re
        return request
    ...
```

分析上述代码如下：

- 调度器中最核心的两个方法是enqueue_request和next_request，它们分别对应请求的入队和出队操作。Spider提交的Request对象最终由Scrapy引擎调用enqueue_request添加到请求队列中，Scrapy引擎同时也调用next_request从请求队列中取出请求，送给下载器下载。
- self.queue和self.df分别是请求队列和去重过滤器对象。在enqueue_request方法中，使用去重过滤器的request_seen方法判断request是否重复，即request对应的页面是否已经爬取过，如果用户没有强制忽略过滤，并且request是重复的，

就抛弃该request，并直接返回False，否则调用self.queue的push方法将request入队，返回True。在next_request方法中，调用self.queue的pop方法出队一个request并返回。

再来看一下创建请求队列和去重过滤器对象的相关代码：

```
import importlib
import six


from scrapy.utils.misc import load_object
from . import connection, defaults


class Scheduler(object):
   def __init__(self, server, ...,
                queue_cls=defaults.SCHEDULER_QUEUE_
                dupefilter_cls=defaults.SCHEDULER_D
       ...
       self.server = server
       self.queue_cls = queue_cls
       self.dupefilter_cls = dupefilter_cls
       ...


   @classmethod
   def from_settings(cls, settings):
       ...
       # 从配置中读取请求队列和去重过滤器的类名
       optional = {
           ...
       'queue_cls': 'SCHEDULER_QUEUE_CLASS',
       'dupefilter_cls': 'DUPEFILTER_CLASS',
       ...
   }
   for name, setting_name in optional.items():
       val = settings.get(setting_name)
       if val:
```

```
            kwargs[name] = val
        ...
        server = connection.from_settings(settings)
        server.ping()

        return cls(server=server, **kwargs)
    ...

    def open(self, spider):
        ...
        # 初始化请求队列
        try:
            self.queue = load_object(self.queue_cls)(
                server=self.server,
                spider=spider,
                key=self.queue_key % {'spider': spider.n
                serializer=self.serializer,
            )
        except TypeError as e:
            raise ValueError("Failed to instantiate que
                        self.queue_cls, e)

        # 初始化去重过滤器
        try:
            self.df = load_object(self.dupefilter_cls)(
                server=self.server,
                key=self.dupefilter_key % {'spider': spi
                debug=spider.settings.getbool('DUPEFILTE
            )
        except TypeError as e:
            raise ValueError("Failed to instantiate dup
             self.dupefilter_cls, e)
        ...
    ...
```

self.queue和self.df的创建是在open方法中调用load_object方法完

成的，load_object方法的参数是类的导入路径（如scrapy_redis.queue.PriorityQueue），这种实现的好处是用户可以使用字符串在配置文件中灵活指定想要使用的队列类和过滤器类。self.queue_cls和self.dupefilter_cls便是从配置文件中读取的导入路径（或默认值）。

2．请求队列的实现

接下来看基于Redis的请求队列的实现。在queque.py中，包含以下3种请求队列：

- PriorityQueue优先级队列（默认）
- FifoQueue先进先出队列
- LifoQueue后进先出队列

我们分析其中代码简短且容易理解的FifoQueue，代码如下：

```
class FifoQueue(Base):
    """Per-spider FIFO queue"""


    def __len__(self):
        """Return the length of the queue"""
        return self.server.llen(self.key)


    def push(self, request):
        """Push a request"""
        self.server.lpush(self.key, self._encode_re


    def pop(self, timeout=0):
        """Pop a request"""
        if timeout > 0:
            # brpop: 阻塞的rpop，可以设置超时
            data = self.server.brpop(self.key, time
            if isinstance(data, tuple):
```

```
                data = data[1]
        else:
            data = self.server.rpop(self.key)
        if data:
            return self._decode_request(data)
```

- self.server是Redis数据库的连接对象（可理解为self.server = redis.StrictRedis(...)），该连接对象是在Scheduler的from_settings方法中创建的，在创建请求队列对象时，被传递给请求队列类的构造器。
- 观察在self.server上调用的方法可知，FifoQueue使用Redis中的一个列表实现队列，该列表在数据库中的键为self.key的值，可以通过配置文件设置（SCHEDULER_QUEUE_KEY），默认为＜spider_name＞:requests。
- push方法对应请求的入队操作，先调用基类的_encode_request方法对request进行编码，然后调用Redis的lpush命令将其插入数据库中列表的最左端（入队）。
- pop方法对应请求的出队操作，调用Redis的rpop或brpop命令从数据库中列表的最右端弹出一个经过编码的request（出队），再调用基类的_decode_request方法对其进行解码，然后返回。
- __len__方法调用Redis的llen命令获取数据库中列表的长度，即请求队列的长度。

下面是FifoQueue、LifoQueue、PriorityQueue共同基类Base的部分代码：

```
from . import picklecompat


class Base(object):
```

```
    """Per-spider base queue class"""


    def __init__(self, server, spider, key, seriali
        """Initialize per-spider redis queue.


        Parameters
        ----------
        server : StrictRedis
            Redis client instance.
        spider : Spider
            Scrapy spider instance.
        key: str
            Redis key where to put and get messages
        serializer : object
            Serializer object with ``loads`` and ``
        """


        if serializer is None:
            serializer = picklecompat
        ...
    self.server = server
    self.spider = spider
    self.key = key % {'spider': spider.name}
    self.serializer = serializer


def _encode_request(self, request):
    """Encode a request object"""
    obj = request_to_dict(request, self.spider)
    return self.serializer.dumps(obj)


def _decode_request(self, encoded_request):
    """Decode an request previously encoded"""
    obj = self.serializer.loads(encoded_request)
    return request_from_dict(obj, self.spider)
...
```

可以看到，在对request进行编、解码时，调用的是self.serializer的dumps和loads方法。self.serializer同样可以通过配置文件指定（SCHEDULER_SERIALIZER），默认为Python标准库中的pickle模块。

3．去重过滤器的实现

最后来看基于Redis的去重过滤器RFPDupeFilter的实现，它位于dupefilter.py中，部分代码如下：

```
...
from scrapy.dupefilters import BaseDupeFilter
from scrapy.utils.request import request_fingerpri
...
class RFPDupeFilter(BaseDupeFilter):
    ...
    def __init__(self, server, key, debug=False):
        """Initialize the duplicates filter.


        Parameters
        ----------
        server : redis.StrictRedis
            The redis server instance.
        key : str
        Redis key Where to store fingerprints.
    debug : bool, optional
        Whether to log filtered requests.


    """
    self.server = server
    self.key = key
    self.debug = debug
    self.logdupes = True


...
```

```
    def request_seen(self, request):
        fp = self.request_fingerprint(request)
        # This returns the number of values added, zero
        added = self.server.sadd(self.key, fp)
        return added == 0


    def request_fingerprint(self, request):
        return request_fingerprint(request)
    ...
```

- self.server是Redis数据库的连接对象，与FifoQueue中相同。
- 观察在self.server上调用的方法可知，RFPDupeFilter使用Redis中的一个集合对请求进行去重，该集合在数据库中的键为self.key的值，可以通过配置文件设置（SCHEDULER_DUPEFILTER_KEY），默认为＜spider_name＞:dupefilter。
- request_fingerprint方法用来获取一个请求的指纹，即请求的唯一标识，请求的指纹是使用Python标准库hashlib中的sha1算法计算得到的（详见scrapy.utils.request中的request_fingerprint函数）。
- request_seen方法用来判断一个请求是否是重复的，先调用request_fingerprint方法计算request的指纹，然后调用Redis的sadd命令尝试将指纹添加到数据库中的集合中，根据sadd返回值判断请求是否重复，返回相应的布尔值结果，即重复返回True，否则返回False。

14.2.2　汇总爬取数据部分

1．RedisPipeline的实现

在分布式爬虫框架中，各个主机爬取到的数据最终要汇总到一处，通常是某种数据库。scrapy-redis提供了一个Item Pipeline(RedisPipeline)，用于将各个爬虫爬取到的数据存入同一个

Redis数据库中。

RedisPipeline位于pipeline.py中，代码如下：

```
from scrapy.utils.misc import load_object
from scrapy.utils.serialize import ScrapyJSONEncod
from twisted.internet.threads import deferToThread


from . import connection, defaults


default_serialize = ScrapyJSONEncoder().encode


class RedisPipeline(object):
    """Pushes serialized item into a redis list/que


    Settings
    --------
    REDIS_ITEMS_KEY : str
        Redis key where to store items.
    REDIS_ITEMS_SERIALIZER : str
        Object path to serializer function.


    """


    def __init__(self, server,
                 key=defaults.PIPELINE_KEY,
                 serialize_func=default_serialize):
        """Initialize pipeline.


        Parameters
        ----------
        server : StrictRedis
```

```
            Redis client instance.
        key : str
            Redis key where to store items.
        serialize_func : callable
            Items serializer function.


        """
        self.server = server
    self.key = key
    self.serialize = serialize_func


@classmethod
def from_settings(cls, settings):
    params = {
        'server': connection.from_settings(settings
    }
    if settings.get('REDIS_ITEMS_KEY'):
        params['key'] = settings['REDIS_ITEMS_KEY']
    if settings.get('REDIS_ITEMS_SERIALIZER'):
        params['serialize_func'] = load_object(
           settings['REDIS_ITEMS_SERIALIZER']
        )


    return cls(**params)


@classmethod
def from_crawler(cls, crawler):
    return cls.from_settings(crawler.settings)


def process_item(self, item, spider):
    return deferToThread(self._process_item, item,


def _process_item(self, item, spider):
    key = self.item_key(item, spider)
```

```
        data = self.serialize(item)
        self.server.rpush(key, data)
        return item


    def item_key(self, item, spider):
        """Returns redis key based on given spider.


        Override this function to use a different key d
        and/or spider.


        """
        return self.key % {'spider': spider.name}
```

- self.server是Redis数据库的连接对象，与FifoQueue中相同。
- 观察在self.server上调用的方法可知，RedisPipeline使用Redis中的一个列表存储所有爬虫爬取到的数据，该列表在数据库中的键为调用item_key方法的结果，即self.key %{'spider': spider.name}。self.key可以通过配置文件设置（REDIS_ITEMS_KEY），默认情况下列表的键为<spider_name>:items。
- Redis的列表只能存储字符串，而Spider爬取到的数据item的类型是Item或Python字典，所以先要将item串行化成字符串，再存入Redis列表。串行化函数也可以通过配置文件指定（REDIS_ITEMS_SERIALIZER），默认情况下item被串行化成json串。
- process_item方法处理爬取到的每一项数据，因为写入数据库为I/O操作，速度较慢，所以可以在线程中执行，调用twisted中的deferToThread方法，启动线程执行_process_item方法。
- _process_item方法实际处理爬取到的每一项数据，先使用self.serial函数将item串行化成字符串，再调用Redis的rpush

命令将其写入数据库中的列表。

14.3　使用scrapy-redis进行分布式爬取

了解了scrapy-redis的原理后，我们学习使用scrapy + scrapy-redis进行分布式爬取。

14.3.1　搭建环境

首先搭建scrapy-redis分布式爬虫环境，当前我们有3台Linux主机。

云服务器（A）：116.29.35.201 (Redis Server)

云服务器（B）：123.59.45.155

本机（C）：1.13.41.127

在3台主机上安装scrapy和scrapy-redis：

```
$ pip install scrapy
$ pip install scrapy-redis
```

选择其中一台云服务器搭建供所有爬虫使用的Redis数据库，步骤如下：

步骤　**01**　在云服务器上安装redis-server。

步骤　**02**　在Redis配置文件中修改服务器的绑定地址（确保数据库可被所有爬虫访问）。

步骤　**03**　启动/重启Redis服务器。

登录云服务器（A），在bash中完成上述步骤：

```
116.29.35.201$ sudo apt-get install redis-server
116.29.35.201$ sudo vi /etc/redis/redis.conf
```

```
...
# bind 127.0.0.1
bind 0.0.0.0
...
116.29.35.201$ sudo service redis-server restart
```

最后，在3台主机上测试能否访问云服务器（A）上的Redis数据库：

```
$ redis-cli -h 116.29.35.201 ping
PONG
```

到此，Scrapy分布式爬虫环境搭建完毕。

14.3.2 项目实战

本章的核心知识点是分布式爬取，因此本项目不再对分析页面、编写Spider等大家熟知的技术进行展示。我们可以任意挑选一个在之前章节中做过的项目，将其改为分布式爬取的版本，这里以第8章的toscrape_book项目（爬取books.toscrape.com中的书籍信息）为例进行讲解。

复制toscrape_book项目，得到新项目toscrape_book_distributed：

```
$ cp -r toscrape_book toscrape_book_distributed
$ cd toscrape_book_distributed
```

在配置文件settings.py中添加scrapy-redis的相关配置：

```
# 必选项
# ==================================================
# 指定爬虫所使用的Redis数据库（在云服务器116.29.35.201
```

```
REDIS_URL = 'redis://116.29.35.201:6379'


# 使用scrapy_redis的调度器替代Scrapy 原版调度器
SCHEDULER = "scrapy_redis.scheduler.Scheduler"


# 使用scrapy_redis的RFPDupeFilter作为去重过滤器
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDup
# 启用scrapy_redis的RedisPipeline将爬取到的数据汇总到R
ITEM_PIPELINES = {
   'scrapy_redis.pipelines.RedisPipeline': 300,
}
# 可选项
# ================================================
#爬虫停止后，保留／清理Redis中的请求队列以及去重集合
# True：保留，False：清理，默认为False
SCHEDULER_PERSIST = True
```

将单机版本的Spider改为分布式版本的Spider，只需做如下简单改动：

```
from scrapy_redis.spiders import RedisSpider


# 1.更改基类
# class BooksSpider(spider.Spider):
class BooksSpider(RedisSpider):
   ...
   # 2.注释start_urls
   #start_urls = ['http://books.toscrape.com/']
   ...
```

上述改动针对“如何为多个爬虫设置起始爬取点”这个问题，解释如下：

- 在分布式爬取时，所有主机上的代码是相同的，如果使用之

前单机版本的Spider代码，那么每一台主机上的Spider都通过start_urls属性定义了起始爬取点，在构造起始爬取点的Request对象时，dont_filter参数设置为了True，即忽略去重过滤器的过滤。因此多个（数量等于爬虫数量）重复请求将强行进入Redis中的请求队列，这可能导致爬取到重复数据。

- 为了解决上述问题，scrapy-redis提供了一个新的Spider基类RedisSpider，RedisSpider重写了start_requests方法，它尝试从Redis数据库的某个特定列表中获取起始爬取点，并构造Request对象（dont_filter＝False），该列表的键可通过配置文件设置（REDIS_START_URLS_KEY），默认为＜spider_name＞:start_urls。在分布式爬取时，用户运行所有爬虫后，需手动使用Redis命令向该列表添加起始爬取点，之后只有其中一个爬虫能获取到起始爬取点，因此对应的请求也就只有一个，从而避免了重复。

到此，分布式版本的项目代码已经完成了，分发到各个主机：

```
$ scp -r toscrape_book_distributed liushuo@116.29.
$ scp -r toscrape_book_distributed liushuo@123.59.
```

分别在3台主机使用相同命令运行爬虫：

```
    $ scrapy crawl books
    2017-05-14 17:56:42 [scrapy.utils.log] INFO: S
    2017-05-14 17:56:42 [scrapy.utils.log] INFO: O
'scrapy_redis.dupefilter.RFPDupeFilter', 'FEED_EXP
'review_rating', 'review_num'], 'SCHEDULER': 'scra
'toscrape_book', 'ROBOTSTXT_OBEY': True, 'NEWSPIDE
'SPIDER_MODULES': ['toscrape_book.spiders']}
    2017-05-14 17:56:42 [scrapy.middleware] INFO:
    ['scrapy.extensions.logstats.LogStats',
     'scrapy.extensions.telnet.TelnetConsole',
```

```
     'scrapy.extensions.corestats.CoreStats']
    2017-05-14 17:56:42 [books] INFO: Reading star
16, encoding: utf-8
    2017-05-14 17:56:42 [scrapy.middleware] INFO:
    ['scrapy.downloadermiddlewares.robotstxt.Robot
     'scrapy.downloadermiddlewares.httpauth.HttpAu
     'scrapy.downloadermiddlewares.downloadtimeout
     'scrapy.downloadermiddlewares.defaultheaders.
     'scrapy.downloadermiddlewares.useragent.UserA
     'scrapy.downloadermiddlewares.retry.RetryMidd
     'scrapy.downloadermiddlewares.redirect.MetaRe
     'scrapy.downloadermiddlewares.httpcompression
     'scrapy.downloadermiddlewares.redirect.Redire
     'scrapy.downloadermiddlewares.cookies.Cookies
     'scrapy.downloadermiddlewares.stats.Downloade
    2017-05-14 17:56:42 [scrapy.middleware] INFO:
    ['scrapy.spidermiddlewares.httperror.HttpError
     'scrapy.spidermiddlewares.offsite.OffsiteMidd
     'scrapy.spidermiddlewares.referer.RefererMidd
     'scrapy.spidermiddlewares.urllength.UrlLength
     'scrapy.spidermiddlewares.depth.DepthMiddlewa
    2017-05-14 17:56:42 [scrapy.middleware] INFO:
     ['scrapy_redis.pipelines.RedisPipeline']
     2017-05-14 17:56:42 [scrapy.core.engine] INFO
     2017-05-14 17:56:42 [scrapy.extensions.logsta
items (at 0 items/min)
     2017-05-14 17:56:42 [scrapy.extensions.telnet
     ...阻塞在此处...
```

运行后，由于Redis中的起始爬取点列表和请求队列都是空的，3个爬虫都进入了暂停等待的状态，因此在任意主机上使用Redis客户端设置起始爬取点：

```
$ redis-cli -h 116.29.35.201
116.29.35.201:6379> lpush books:start_urls 'http:/
(integer) 1
```

随后，其中一个爬虫（本实验中是云服务器A）从起始爬取点列表中获取到了url，在其log中观察到如下信息：

```
2017-05-14 17:57:18 [books] DEBUG: Read 1 requests
```

该爬虫用起始爬取点url构造的Request对象最终被添加到Redis中的请求队列之后。各个爬虫相继开始工作了，可在各爬虫的log中观察到类似于如下的信息：

```
      2017-05-14 18:00:42 [scrapy.core.scraper] DEB
http://books.toscrape.com/catalogue/arena_587/inde
      {'name': 'Arena',
       'price': '£21.36',
       'review_num': '0',
       'review_rating': 'Four',
       'stock': '11',
       'upc': '2c34f9432069b52b'}
      2017-05-14 18:00:42 [scrapy.core.engine] DEBU
http://books.toscrape.com/catalogue/page-21.html)
      2017-05-14 18:00:42 [scrapy.core.scraper] DEB
http://books.toscrape.com/catalogue/adultery_586/i
      {'name': 'Adultery',
       'price': '£20.88',
       'review_num': '0',
       'review_rating': 'Five',
       'stock': '11',
       'upc': 'bb967277222e689c'}
      2017-05-14 18:00:42 [scrapy.core.engine] DEBU
http://books.toscrape.com/catalogue/page-21.html)
      2017-05-14 18:00:42 [scrapy.core.scraper] DEB
http://books.toscrape.com/catalogue/a-mothers-reck
l>
      {'name': "A Mother's Reckoning: Living in the
       'price': '£19.53',
       'review_num': '0',
       'review_rating': 'Three',
```

```
            'stock': '11',
            'upc': '2b69dec0193511d9'}
          2017-05-14 18:00:43 [scrapy.core.scraper] DEB
http://books.toscrape.com/catalogue/112263_583/ind
          {'name': '11/22/63',
            'price': '£48.48',
            'review_num': '0',
            'review_rating': 'Three',
            'stock': '11',
            'upc': 'a9d7b75461084a26'}
          2017-05-14 18:00:43 [scrapy.core.engine] DEBU
http://books.toscrape.com/catalogue/page-21.html)
          2017-05-14 18:00:43 [scrapy.core.scraper] DEB
http://books.toscrape.com/catalogue/10-happier-how
losing-my-edge-and-found-self-help-that-actually-w
          {'name': '10% Happier: How I Tamed the Voice
                   'Without Losing My Edge, and Found Se
            'price': '£24.57',
            'review_num': '0',
            'review_rating': 'Two',
            'stock': '10',
            'upc': '34669b2e9d407d3a'}
```

等待全部爬取完成后，在Redis中查看爬取到的数据：

```
116.29.35.201:6379> keys *
1) "books:items"
2) "books:dupefilter"
116.29.35.201:6379> llen books:items
(integer) 1000
116.29.35.201:6379> LRANGE books:items 0 4
          1) "{\"stock\": \"22\", \"review_num\": \"0\"
the Attic\", \"review_rating\": \"Three\", \"price
          2) "{\"stock\": \"20\", \"review_num\": \"0\"
Objects\", \"review_rating\": \"Four\", \"price\":
          3) "{\"stock\": \"20\", \"review_num\": \"0\"
Velvet\", \"review_rating\": \"One\", \"price\": \
          4) "{\"stock\": \"20\", \"review_num\": \"0\"
```

```
\"Soumission\", \"review_rating\": \"One\", \"pric
    5) "{\"stock\": \"19\", \"review_num\": \"0\"
Little Secrets of Getting Your Dream Job\", \"revi
    116.29.35.201:6379> LRANGE books:items -5 -1
    1) "{\"name\": \"Shameless\", \"price\": \"\\
\"c068c013d6921fea\", \"review_num\": \"0\", \"sto
    2) "{\"stock\": \"1\", \"review_num\": \"0\",
Devotion (The Regency Spies of London #1)\", \"rev
    3) "{\"stock\": \"1\", \"review_num\": \"0\",
(Women's Murder Club #1)\", \"review_rating\": \"C
    4) "{\"stock\": \"1\", \"review_num\": \"0\",
to See Before You Die\", \"review_rating\": \"Five
    5) "{\"name\": \"Girl in the Blue Coat\", \"p
\"upc\": \"41fc5dce044f16f5\", \"review_num\": \"0
```

如上所示，我们成功地爬取到了1000项数据（由各爬虫最后的log信息得知，爬虫A:514项，爬虫B:123项，爬虫C:363项）。每一项数据以json形式存储在Redis的列表中，需要使用这些数据时，可以编写Python程序将它们从Redis中读出，代码框架如下：

```
import redis
import json


ITEM_KEY = 'books:items'


def process_item(item):
    # 添加处理数据的代码
    ...


def main():
    r = redis.StrictRedis(host='116.29.35.201', por
    for _ in range(r.llen(ITEM_KEY)):
        data = r.lpop(ITEM_KEY)
        item = json.loads(data.decode('utf8'))
```

```
            process_item(item)


if __name__ == '__main__':
    main()
```

到此，我们完成了分布式爬取的项目。

14.4　本章小结

本章我们学习了如何利用scrapy-redis构建分布式Scrapy爬虫，首先介绍了一些Redis数据库的基础知识，然后对scrapy-redis源码进行了分析，最后通过案例展示了一个分布式爬虫的开发流程。